



US005557798A

United States Patent [19]

Skeen et al.

[11] Patent Number: **5,557,798**[45] Date of Patent: ***Sep. 17, 1996**

[54] **APPARATUS AND METHOD FOR PROVIDING DECOUPLING OF DATA EXCHANGE DETAILS FOR PROVIDING HIGH PERFORMANCE COMMUNICATION BETWEEN SOFTWARE PROCESSES**

[75] Inventors: Marion D. Skeen, Palo Alto; Mark Bowles, Woodside, both of Calif.

[73] Assignee: Tibco, Inc., Palo Alto, Calif.

[*] Notice: The term of this patent shall not extend beyond the expiration date of Pat. No. 5,257,369.

4141689	12/1989	Australia	G06F 13/42
79455/91	6/1991	Australia	G06F 15/16
0108233	5/1984	European Pat. Off.	G06F 15/16
0130375	1/1985	European Pat. Off.	G06F 15/40
0167725	1/1986	European Pat. Off.	G06F 5/00
0216535	8/1986	European Pat. Off.	G06F 15/16
0258867	3/1988	European Pat. Off.	G06F 9/46
0387462	9/1990	European Pat. Off.	G06G 15/21
57-92954	of 1980	Japan	
63-50140	of 1986	Japan	
63-214045	of 1987	Japan	
63-174159	of 1987	Japan	
2191069	12/1987	United Kingdom	G09G 1/00
2205018	11/1988	United Kingdom	G06F 15/40

OTHER PUBLICATIONS

ISIS and the Meta Project; K. Birman and K. Marzullo; published in Sun Technology, Summer 1989.

News Need Not be Slow; G. Collyer and H. Spencer; published in Winter 1987 USENIX Technical Conference; Winter 1987.

(List continued on next page.)

Primary Examiner—Kevin A. Kriess

Attorney, Agent, or Firm—Ronald Fish; Falk, Vestal & Fish

[57] ABSTRACT

A communication interface for decoupling one software application from another software application such communications between applications are facilitated and applications may be developed in modularized fashion. The communication interface is comprised of two libraries of programs. One library manages self-describing forms which contain actual data to be exchanged as well as type information regarding data format and class definition that contain semantic information. Another library manages communications and includes a subject mapper to receive subscription requests regarding a particular subject and map them to particular communication disciplines and to particular services supplying this information. A number of communication disciplines also cooperate with the subject mapper or directly with client applications to manage communications with various other applications using the communication protocols used by those other applications.

56 Claims, 20 Drawing Sheets

[21] Appl. No.: 632,551

[22] Filed: Dec. 21, 1990

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 601,117, Oct. 22, 1990, Pat. No. 5,257,369, which is a continuation-in-part of Ser. No. 386,584, Jul. 27, 1989, Pat. No. 5,187,787.

[51] Int. Cl.⁶ G06F 15/16; G06F 13/00

[52] U.S. Cl. 395/650; 364/280; 364/284; 364/284.3; 364/281.3; 364/DIG. 1

[58] Field of Search 395/650, 700

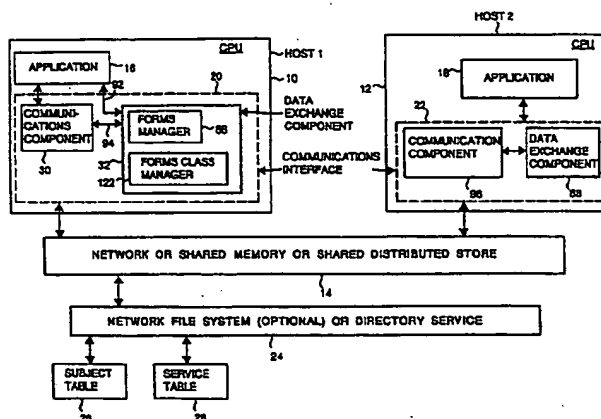
[56] References Cited**U.S. PATENT DOCUMENTS**

4,463,093	12/1982	Davis et al.	364/200
4,688,170	8/1987	Waite et al.	364/200
4,718,005	1/1988	Feigenbaum et al.	364/200
4,751,635	6/1988	Kret	
4,815,030	3/1989	Cross et al.	364/900
4,815,988	3/1989	Trottier et al.	364/200
4,823,122	4/1989	Mann et al.	340/825.28
4,851,988	7/1989	Trottier et al.	364/200

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

41417/89	9/1989	Australia	G06F 15/16
41416/89	9/1989	Australia	G06F 13/42
41417/89	12/1989	Australia	G06F 15/16



U.S. PATENT DOCUMENTS

4,914,583	4/1990	Weisshaar et al.	364/200
4,937,784	6/1990	Masai et al.	364/900
4,975,830	12/1990	Gerpheide et al.	364/200
4,975,904	12/1990	Mann et al.	370/85.1
4,975,905	12/1990	Mann et al.	370/85.1
4,992,972	2/1991	Brooks et al.	364/900
4,999,771	3/1991	Ralph et al.	364/200
5,058,108	10/1991	Mann et al.	370/85.1
5,062,037	10/1991	Shorter et al.	364/200
5,073,852	12/1991	Siegel et al.	395/700
5,101,406	3/1992	Messenger	370/94.1
5,187,787	2/1993	Skeen et al.	395/600
5,257,369	10/1993	Skeen et al.	395/650

OTHER PUBLICATIONS

The USENET System; H. Henderson; 1987.

Integration Mechanisms in the FIELD Environment; S. Reiss; Technical Report No. CS-88-18, published by Department of Computer Science, Brown University, Oct. 1988.

A Retrospective and Evaluation of the Amoeba Distributed Operating System; A. Tanenbaum, R. vanRenesse, H. van-Staveren, and S. Mullender, published in 1988.

Exploiting Virtual Synchrony in Distributed Systems; K. Birman and T. Joseph; Proceedings of the Eleventh ACM Symposium on Operating System Principles; {ACM Press}, New York, N.Y., Nov. 1987; also published as a special issue of Operating Systems Review, a quarterly publication of the ACM.

TIB Reference Manual, "The Teknekron Information Bus™: Programmer's Reference Manual," Version 1.1, Sep. 7, 1989, pp. 1-46.

"BASIS Application Programming Interface (AIP)," pp. 1-82.

"BASIS Objectives, Environments, Concepts Functions, Value for Business Partners and Customers," IBM Confidential.

DataTrade R1, "Lans Lans/Wans," Aug. 23, 1990, pp. 1-4.

DataTrade R1, "Lans DT R1 Software Components," Aug. 23, 1990, pp. 1-7.

DataTrade R1, "Lans DT R1 Network Architecture," Aug. 23, 1990, pp. 1-14.

DataTrade R1, "Lans Broadcast Concepts," Aug. 23, 1990, pp. 1-9.

DataTrade R1, "Lans Broadcast Performance," Aug. 23, 1990, pp. 1-3.

DataTrade R1, "Lans Point-Point Concepts," Aug. 23, 1990, pp. 1-4.

DataTrade R1, "Lans Security," Aug. 23, 1990, pp. 1-4.

DataTrade R1, "API Overview," Jun. 6, 1990, pp. 1-11.

DataTrade R1, "API DataTrade API Verbs," Jun. 6, 1990, pp. 1-14.

DataTrade R1, "DataTrade Using DataTrade: APs," Aug. 23, 1990, pp. 1-14.

"Delivering Integrated Solutions," 6 pages.

Digital, "RAMS Message Bus for VAX/VMS," May 11, 1990, pp. 1-3.

Howard Kilman and Glen Macko, "An Architectural Perspective of a Common Distributed Heterogeneous Message Bus," 1987, pp. 171-184.

Glen Macko, "Developing a Message Bus for Integrating VMS High Speed Task to Task Communications," Fall 1986, pp. 339-347.

Steven G. Judd, "A Practical Approach to Developing Client-Server Applications Among VAX/VMS, CICS/VS, and IMS/VS LU6.2 Applications Made Easy," Spring 1990, pp. 95-112.

Product Insight, "Don't Miss the Lates Message Bus, VAX-PAMSV2.5," Jun. 1989, pp. 18-21.

Digital Equipment Corporation, "Digital Packaged Application Software Description PASD PASD Name: VAX-PAMS PASD: US.002.02," Version 2.5, Dec. 5, 1989, pp. 1-8.

Digital Equipment Corporation, "PAMS Basic Call Set PAMS Message BUS Efficient Task-to-Task Communication," Jul. 1989, pp. 1-25.

Digital Equipment Corporation, "Package Application Software Description for ULTRIX-PAMS," Version 1.2, Dec. 5, 1989, pp. 1-7.

Digital Equipment Corporation, "Package Application Software Description for PC-PAMS," Version 1.2, Dec. 5, 1989, pp. 1-7.

Digital Equipment Corporation, "PAMS Self-Maintenance Service Description," Apr. 3, 1990, pp. 1-3.

Digital Equipment Corporation, "LU6.2 PAMS Self-Maintenance Service Description," Apr. 3, 1990, pp. 1-3.

Digital Equipment Corporation, "PAMS Installation and Orientation Service Description," Jan. 31, 1989, pp. 1-3.

Digital Equipment Corporation, "PAMS LU6.2 Installation and Orientation Service Description," Apr. 19, 1990, pp. 1-3.

Digital Equipment Corporation, "Package Application Software Description for PAMS LU6.2," Version 2.1, Apr. 19, 1990, pp. 1-18.

Carriera and Galernter, "Linda In Context", Communications of the ACM, Apr. 1989, vol. 32, No. 4, pp. 444-458.

IBM DataTrade System introduced Mar. 13, 1990.

Digital Equipment Corporation PAM, Jul. 1991.

Goldman Sachs Development Effort (see Information Disclosure Statement filed with this form, entry #5).

Salomon Brothers Activities (see Information Disclosure Statement filed with this form, entry #6).

The Metamorphosis of Information Management; David Gelernter; Scientific American, Aug. 1989; pp. 66-73.

Schroeder et al., *Experience with Grapevine: The Growth of a Distributed System*, ACM Transactions on Computer Systems, vol. 2, No. 1, Feb. 1984, pp. 3-23.

Cheriton, *Distributed Process Groups in the U Kernel*, ACM Transactions on Computer Systems, vol. 3, No. 2, May 1985, pp. 77-107.

Birman, et al., *ISIS Systems Manual*, Mar. 1988.

"A Stub Generator for Multilanguage RPC in Heterogeneous Environments", P. Gibbons, IEEE Trans. on Software Engineering vol. SE-13, No. 1, Jan. 1987.

Source Code for the Isis file tk-news.c dated May 1990, Feb. 24, 1988 and Dec. 14, 1987.

CCITT Standard X.208.

CCITT Standard X.209.

"Man" pages for Sun Release 4.1, Nov., 1987.

Tanenbaum, Computer Networks (2nd Edition), copyright 1988 by Prentice-Hall, Inc., pp. 475 through 490.

Birman, et al., "The Isis System Manual," 1988, pp. 188-191, Isis Distributed News.

Birman & Joseph, Reliable Communication in the Presence of failures, 1987, ACM Transactions on Computer Systems, vol. 5, No. 1, pp. 47-76.

Birman, et al. Isis Documentation: Release1, 1987, Dept Comp Science, Cornell University p. 30.

Birman, "Exploiting Virtual Synchrony in Distributed Systems" Operating Systems Review, vol. 21, No. 5, Proceeding of 11th ACM Symposium on Operating Systems Principles, Nov. 1987.

Lum, Shu & Housel, "A General Methodology for data Conversion and Restructuring": Sep. 1986 issue Data Conversion, vol. 20, No. 5.

IBM Corp, Technical Disclosure Bulletin, Oct. 1985, G06F15/20 F3C.

Gordon, "Providing Multiple-Channel Communication Using the Experimental Digital Switch." 1982 IEEE Transactions on Communications, vol. COM-30, No. 6.

Hughes, *A Multicast Interface for UNIX 4.3, Software Practice and Experience*, vol. 18(1), 15-27 Jan. 1988.

Frank, et al., *Multicast Communication on Network Computers*, IEEE, Article published in IEEE Software, May 1985, pp. 49-61.

Skeen, et al., *Reliable Message Diffusion*, Draft Oct. 9, 1987.

Oskiewicz, et al., *ISA Project, A Model for Interface Groups*, 1990.

Birman, et al., *Reliable Communication in the Presence of Failures*, ACM Transactions on Computer systems, vol. 5, No. 1, Feb. 1987, pp. 47-76.

French, et al., *The Zephyr Programmer's Manual*, Apr. 5, 1989.

DellaFerra, et al., *The Zephyr Notification Service*, Usenet Conference Feb. 1988.

DellaFerra, et al., *Project Athena Technical Plan*, Section E.4.1, Zephyr Notification Service, Jun. 5, 1989.

Bellville, *Zephyr on Athena*, Sep. 10, 1991, Version 3.

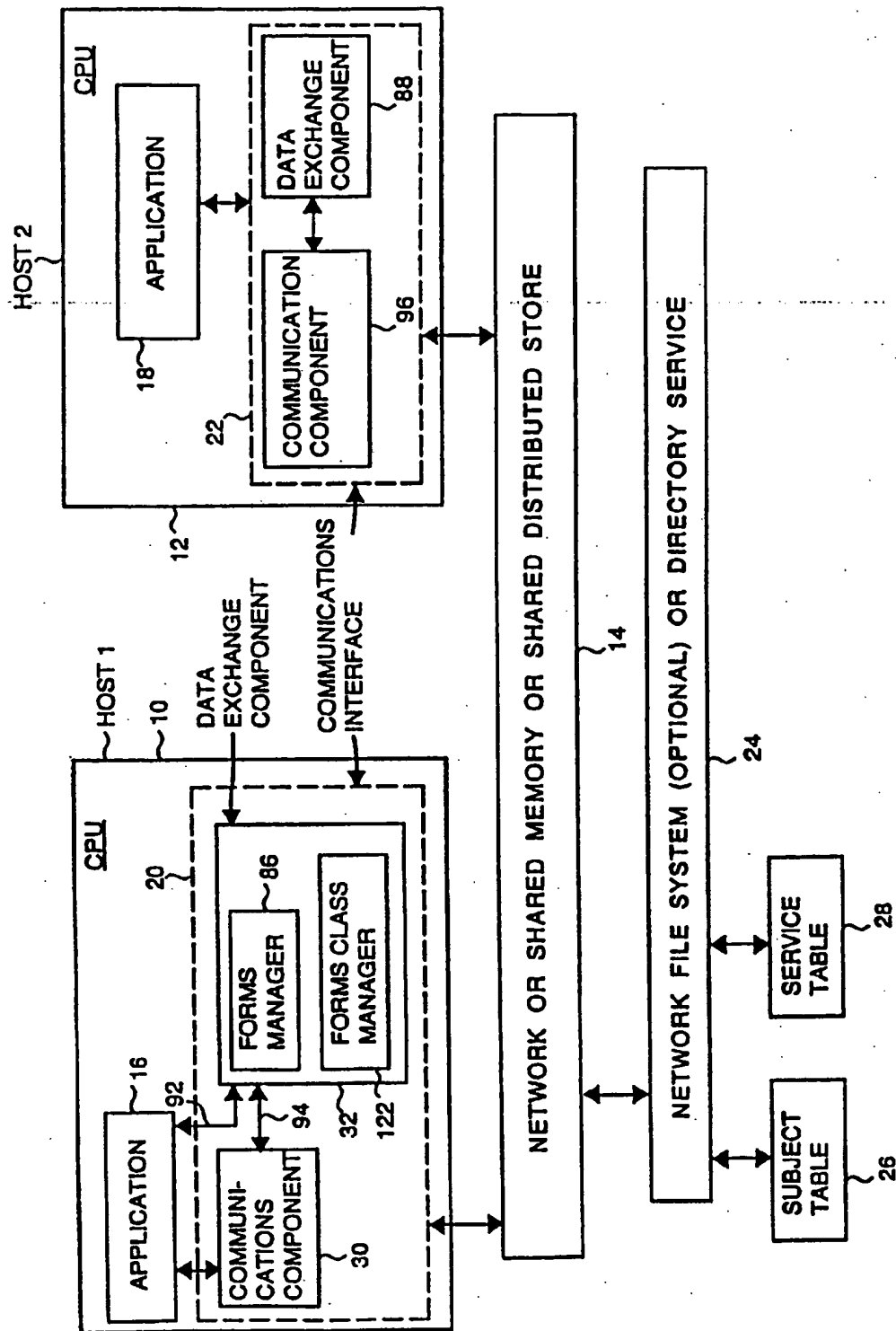


FIGURE 1

EXAMPLE FORM CLASS DEFINITIONS

PLAYER _ NAME: CLASS 1000
RATING: FLOATING _ POINT CLASS 11
AGE: INTEGER CLASS 12
LAST _ NAME: STRING _ 20 _ ASCII CLASS 10
FIRST _ NAME: STRING _ 20 _ ASCII CLASS 10

FIGURE 2

PLAYER _ ADDRESS: CLASS 1001
STREET: STRING _ 20 _ ASCII CLASS 10
CITY: STRING _ 20 _ ASCII CLASS 10
STATE: STRING _ 20 _ ASCII CLASS 10

FIGURE 3

TOURNAMENT _ ENTRY: CLASS 1002
TOURNAMENT _ NAME: STRING _ 20 _ ASCII CLASS 10
PLAYER: PLAYER _ NAME CLASS 1000
ADDRESS: PLAYER _ ADDRESS CLASS 1001

FIGURE 4

STRING _ 20 _ ASCII: CLASS 10
STRING _ 20 ASCII
INTEGER: CLASS 12
INTEGER _ 3
FLOATING - POINT : CLASS 11
FLOATING _ POINT _ 1/1

FIGURE 5

INSTANCE OF FORM OF CLASS TOURNAMENT_ENTRY
CLASS 1002 AS STORED IN MEMORY

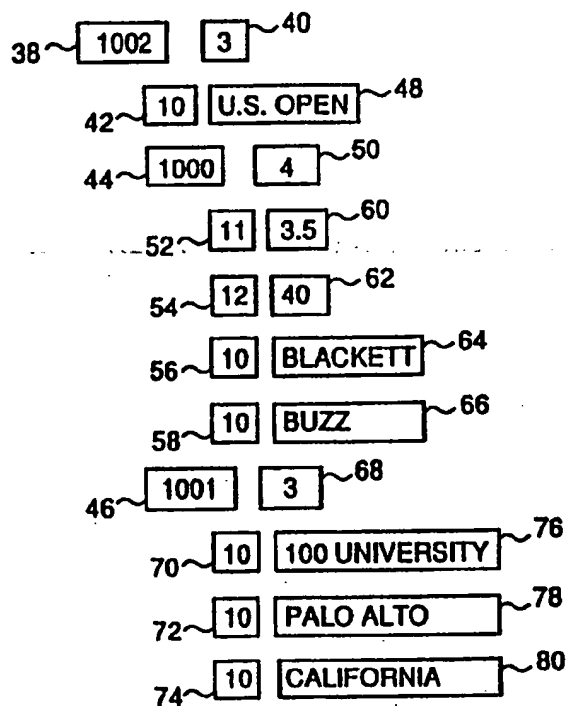


FIGURE 6

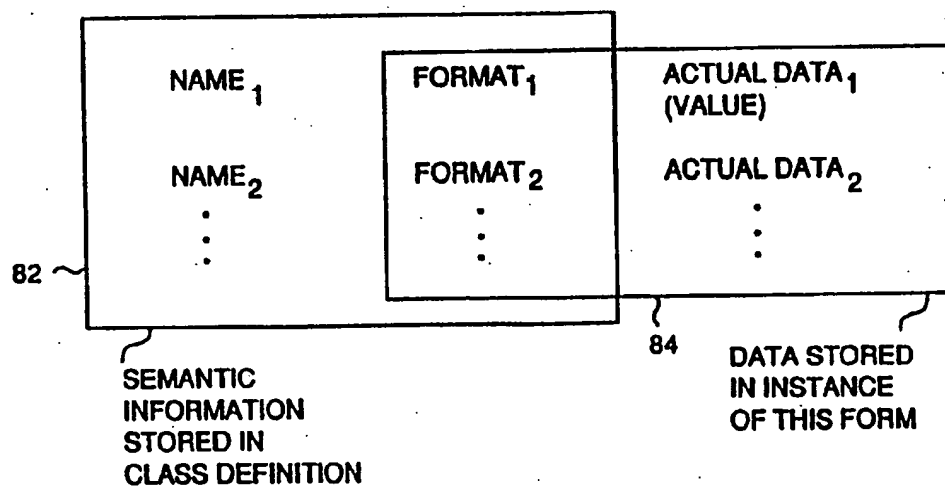
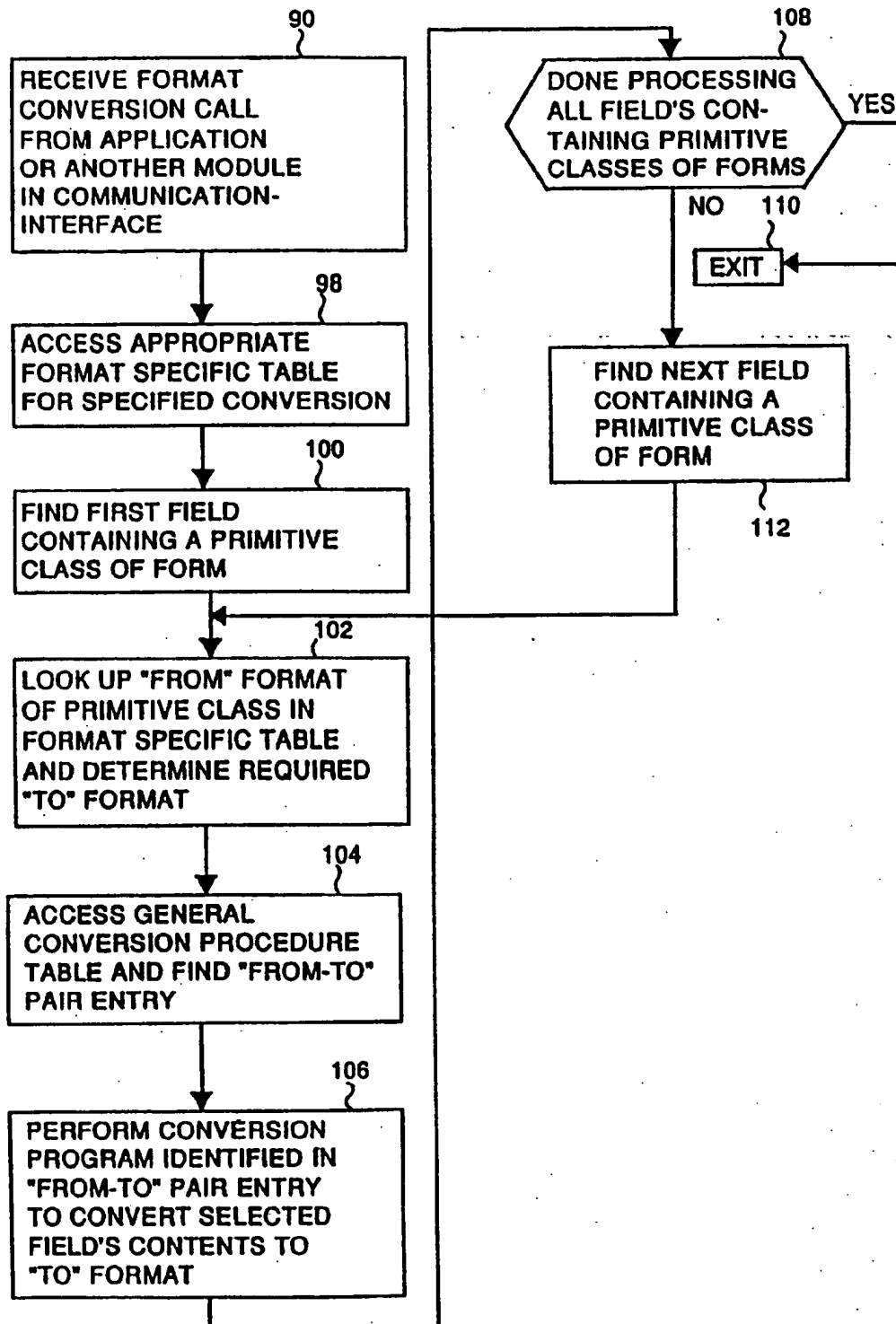


FIGURE 7



FORMAT OPERATION
FIGURE 8

DEC TO
ETHERNET™

FROM	TO
11	15
12	22
10	25
.	
.	
.	

FIGURE 9

ETHERNET™
TO IBM

FROM	TO
15	31
22	33
25	42
.	.
.	.
.	.

FIGURE 10

GENERAL CONVERSION
PROCEDURES TABLE

FROM	TO	CONVERSION PROGRAM
11	15	FLOAT I _ ETHER
12	22	INTEGER I _ ETHER
10	25	ASCII _ ETHER
15	31	ETHER _ FLOAT 2
33	22	ETHER _ INTEGER
42	25	ETHER _ EBCDIC
.	.	.
.	.	.
.	.	.

FIGURE 11

SEMANTIC-DEPENDENT PROCESSING

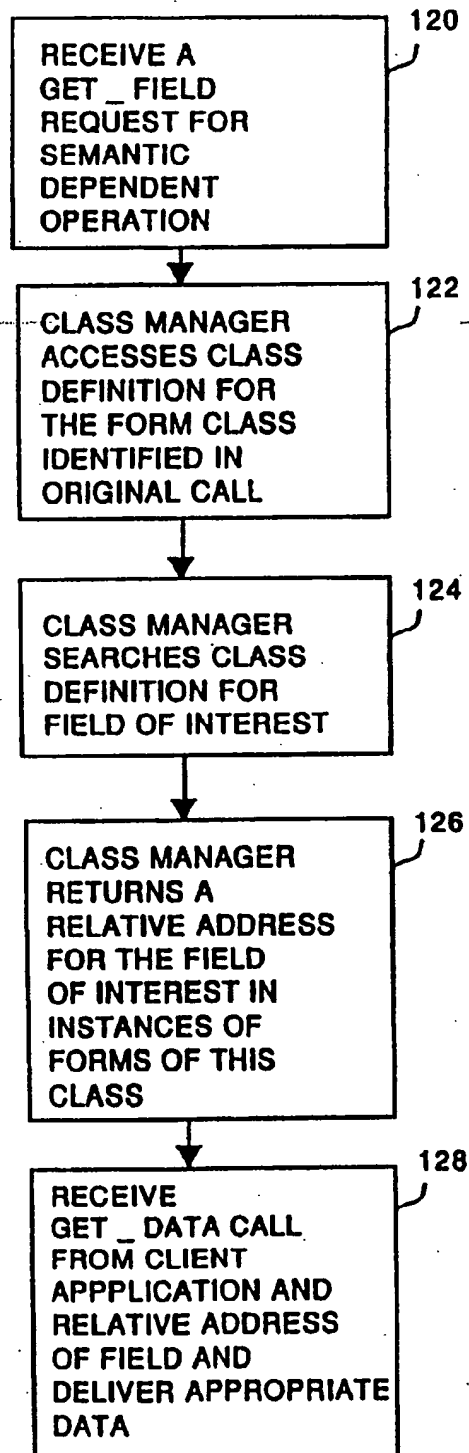


FIGURE 12

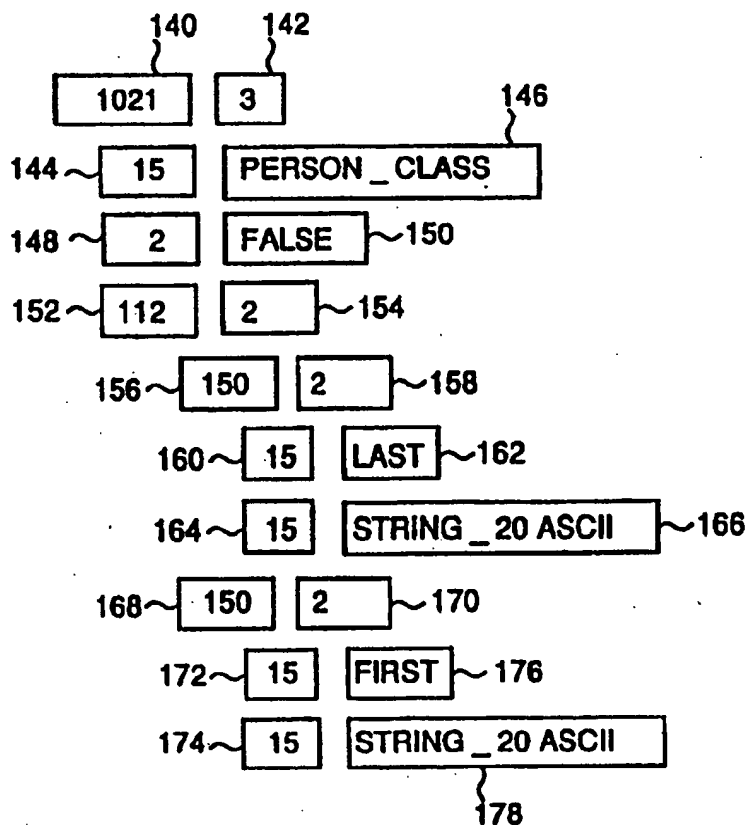
PERSON_CLASS: CLASS 1021

LAST: STRING_20 ASCII

FIRST: STRING_20 ASCII

CLASS DEFINITION

FIGURE 13A



CLASS DESCRIPTOR STORED
IN RAM AS A FORM

FIGURE 13B

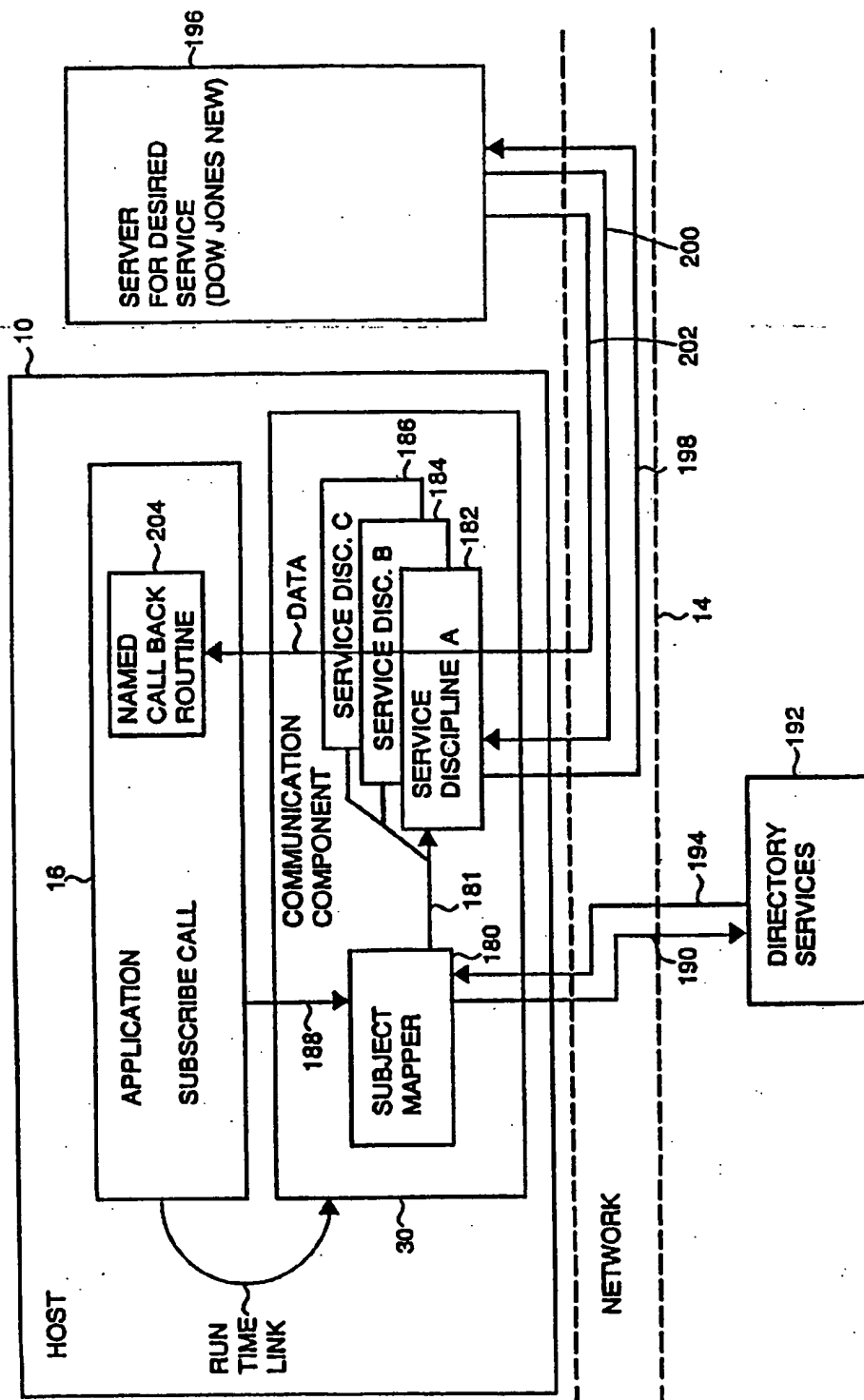
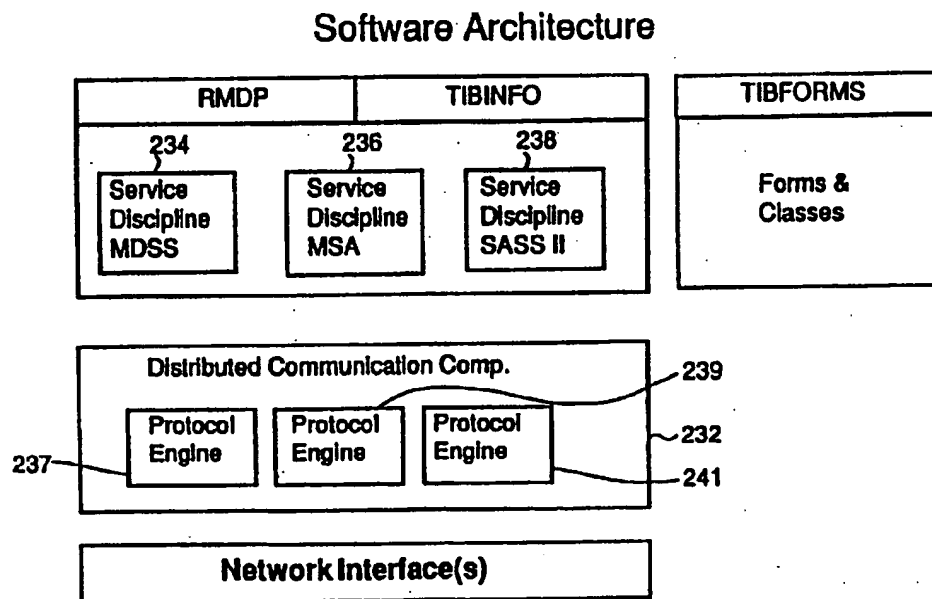
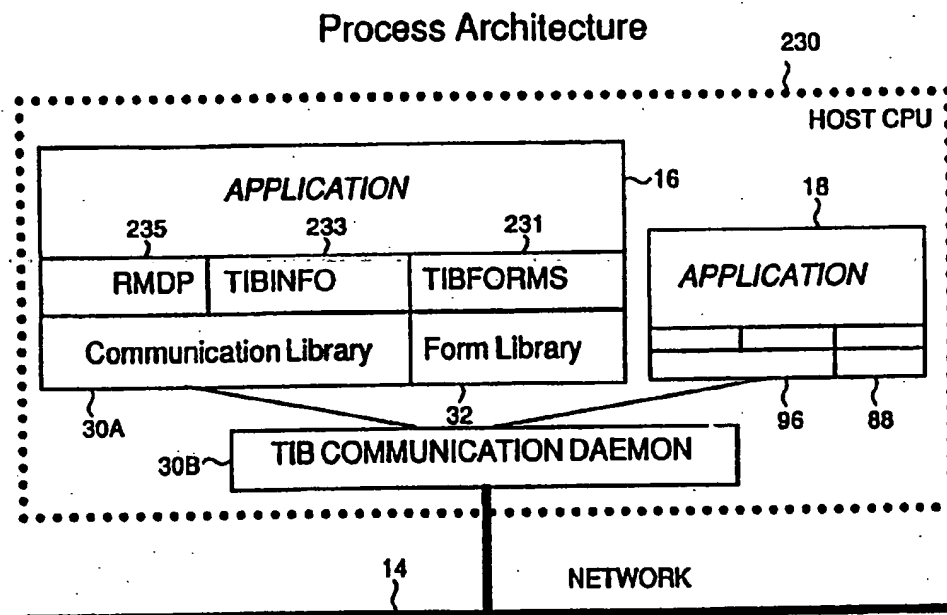


FIGURE 14



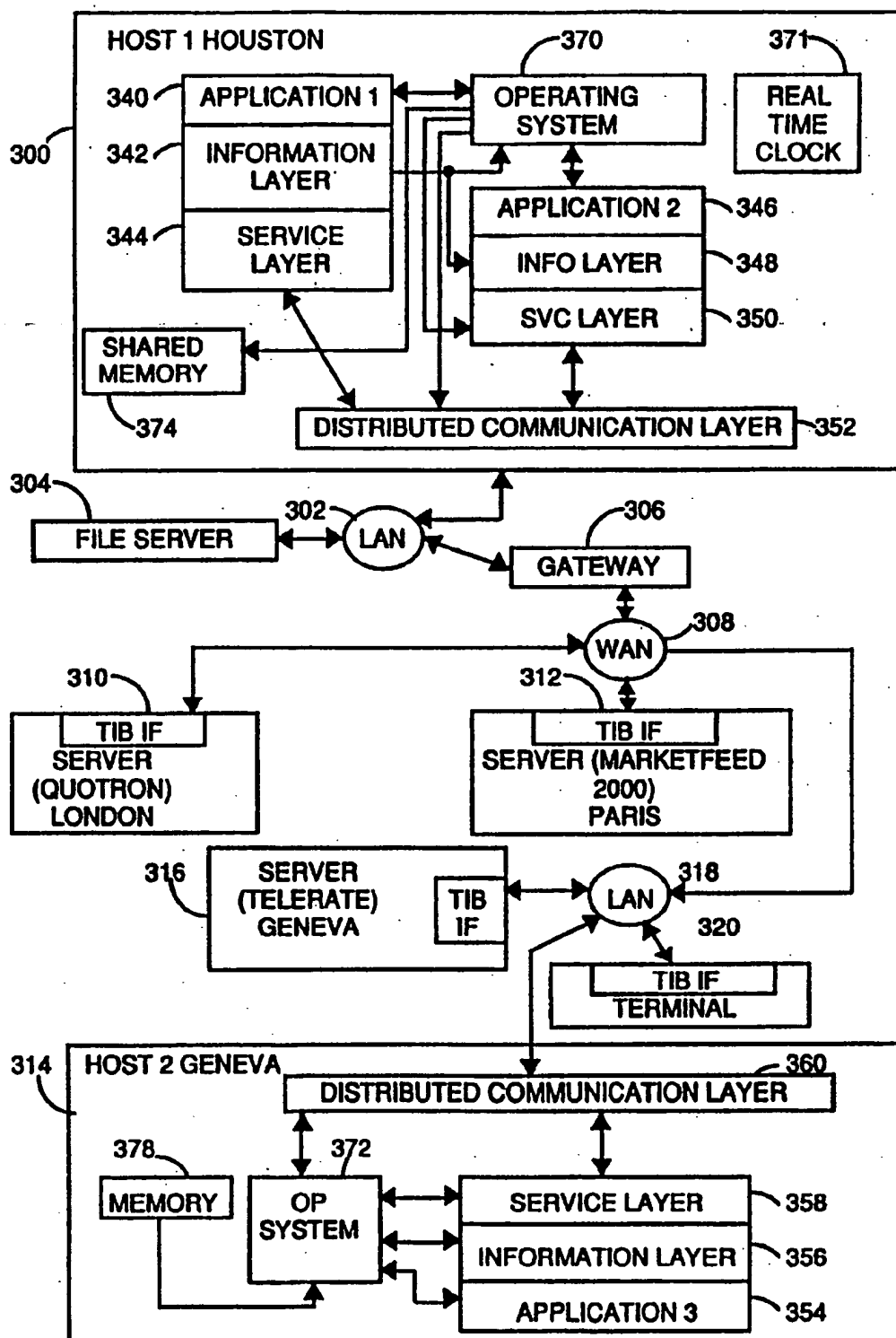
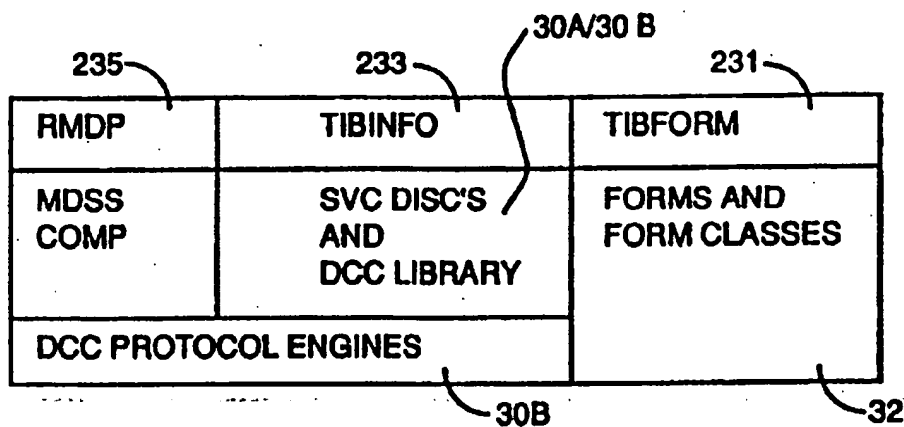
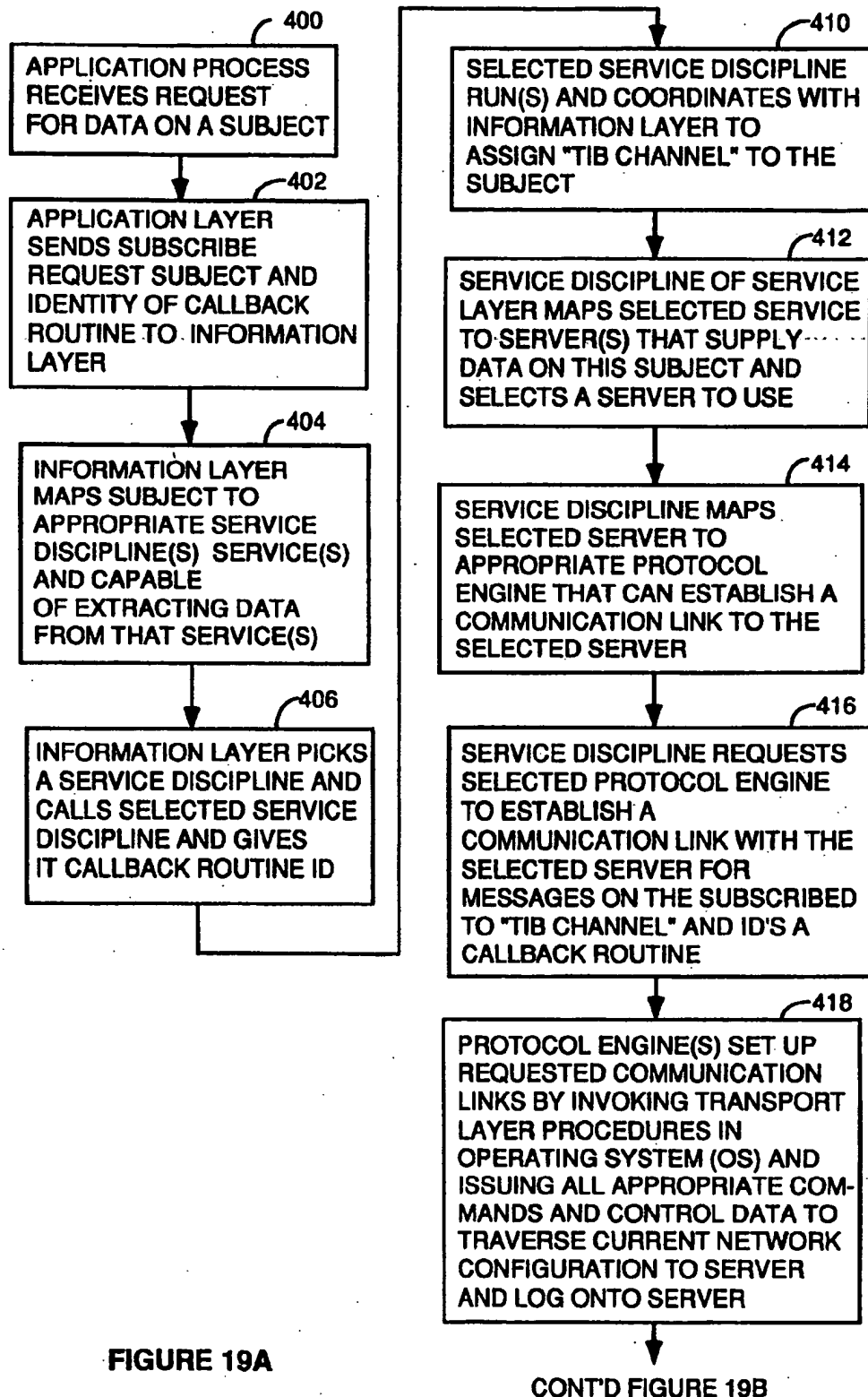


FIGURE 17

**FIGURE 18**



FROM FIGURE 19 A

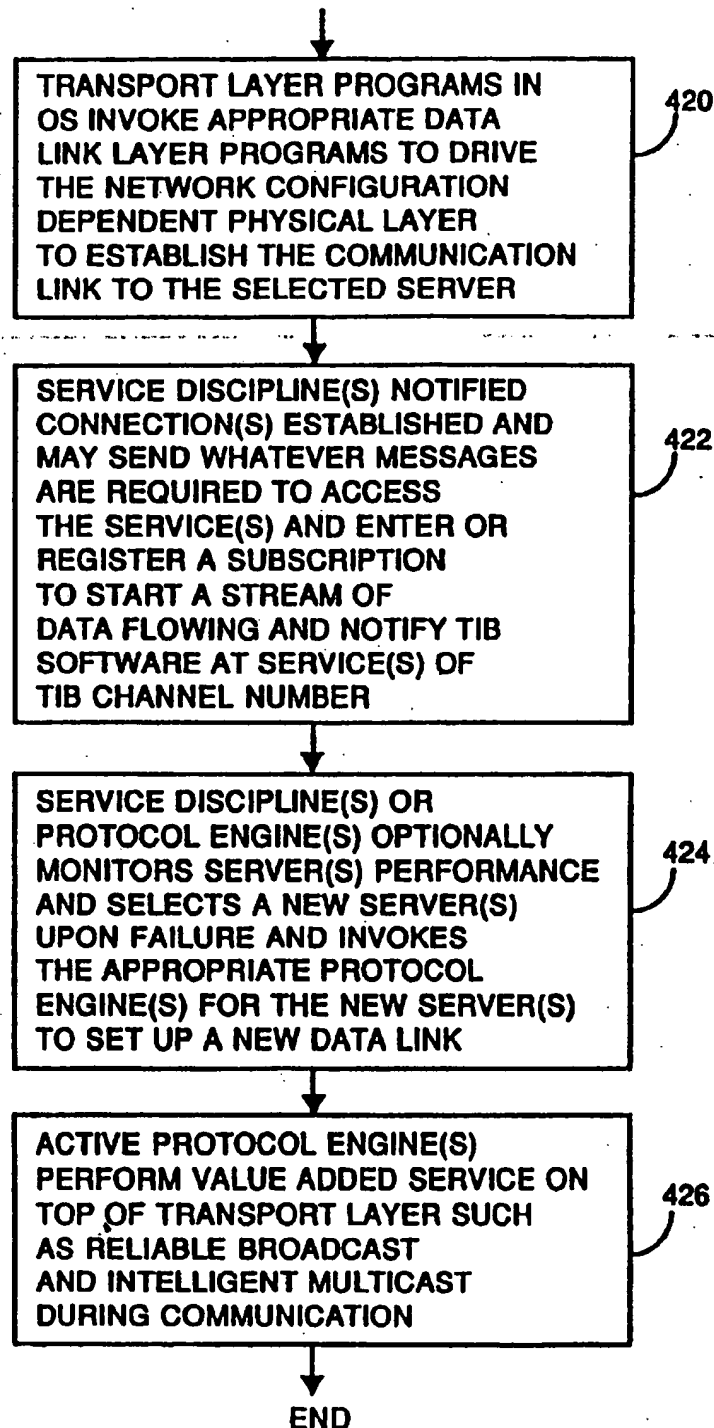


FIGURE 19 B

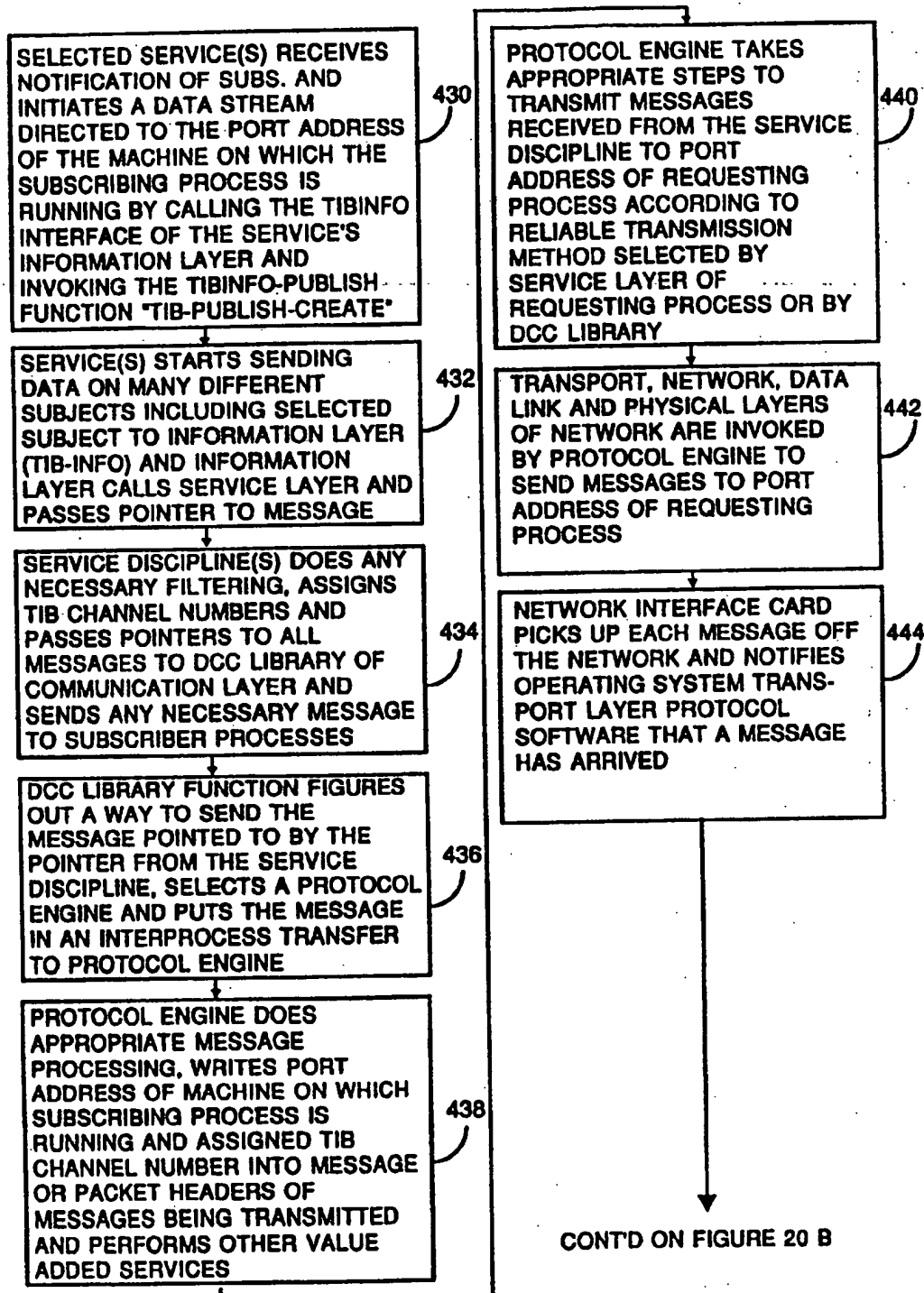


FIGURE 20 A

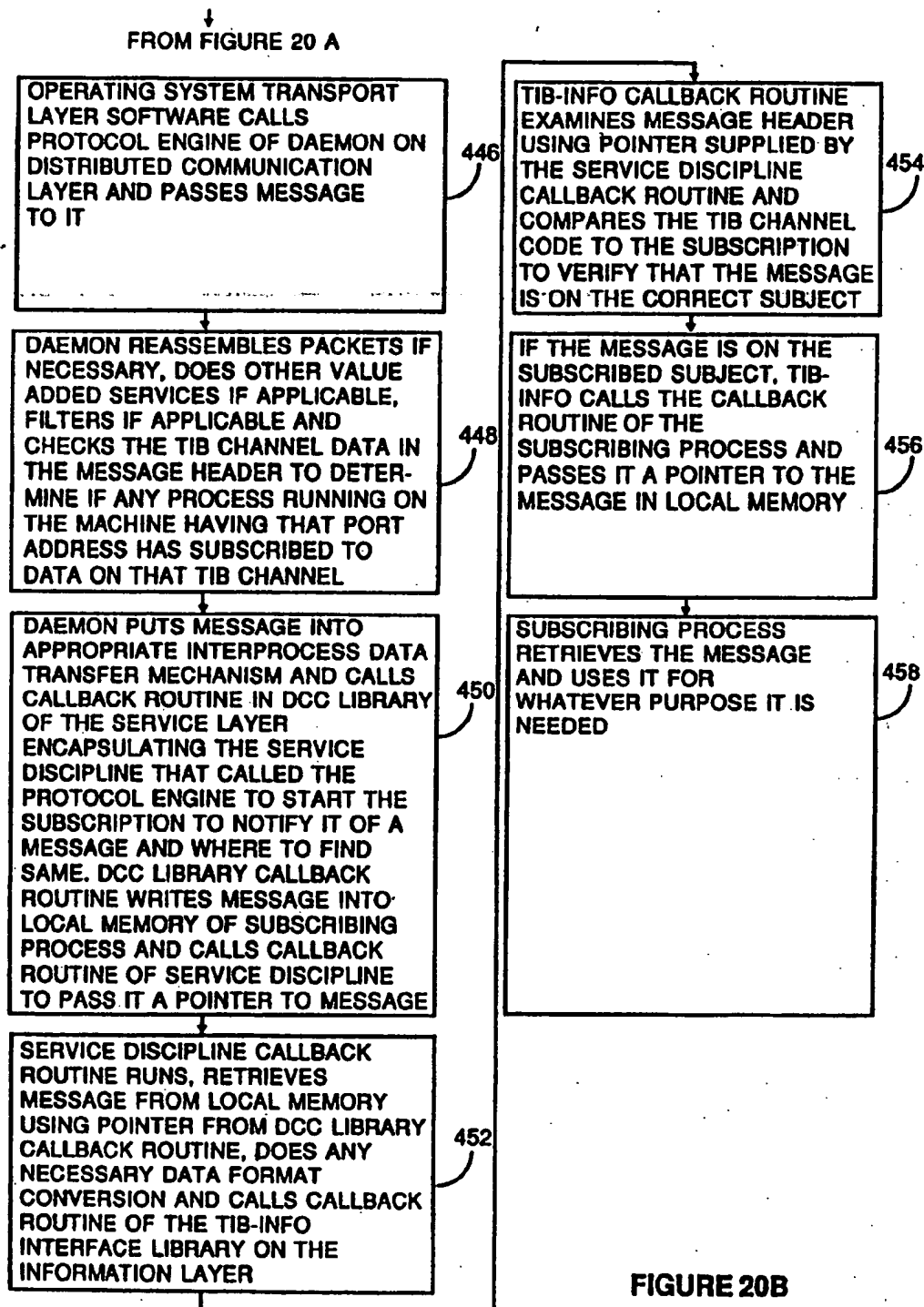


FIGURE 20B

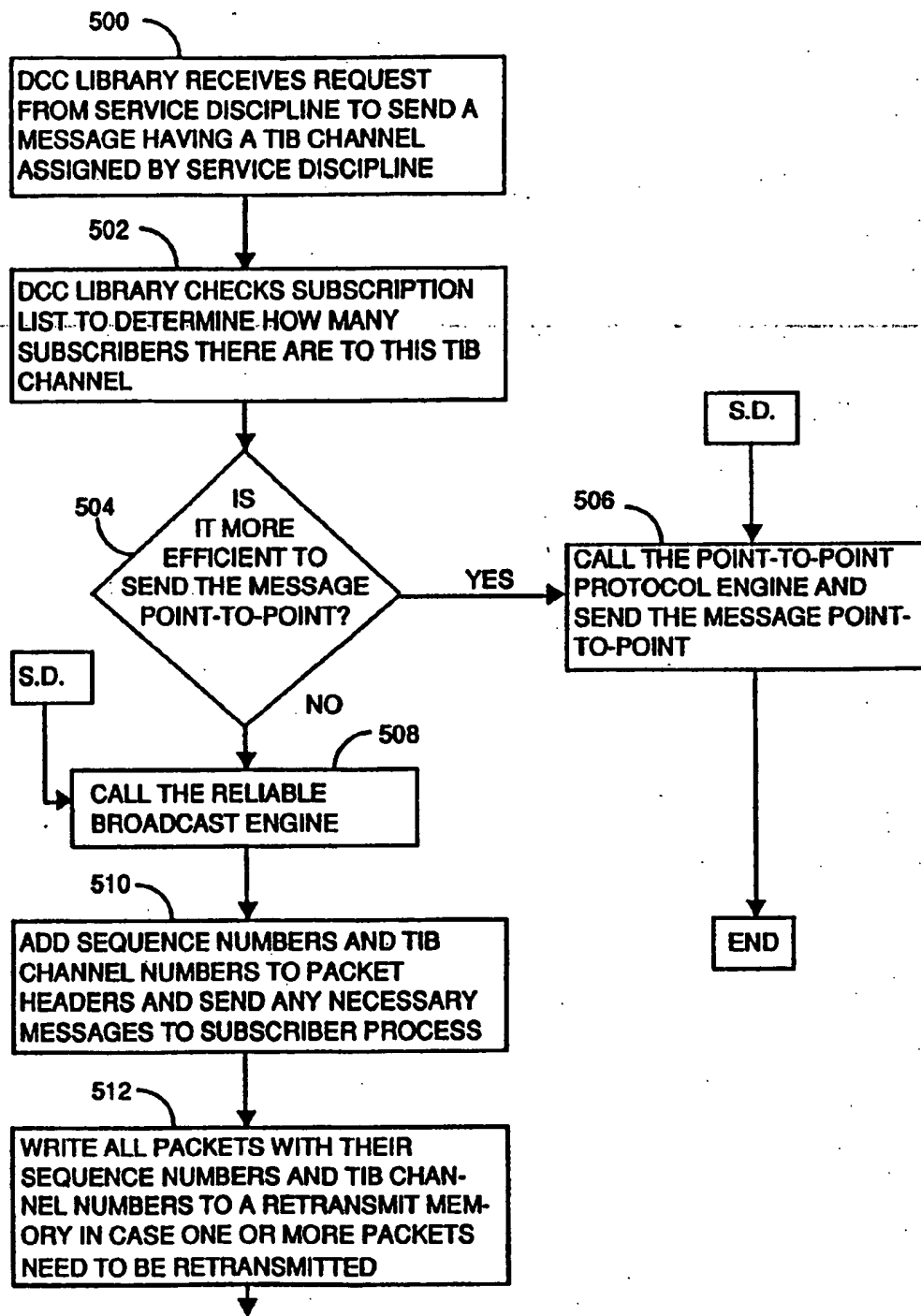


FIGURE 21 A

FROM FIGURE 21 A

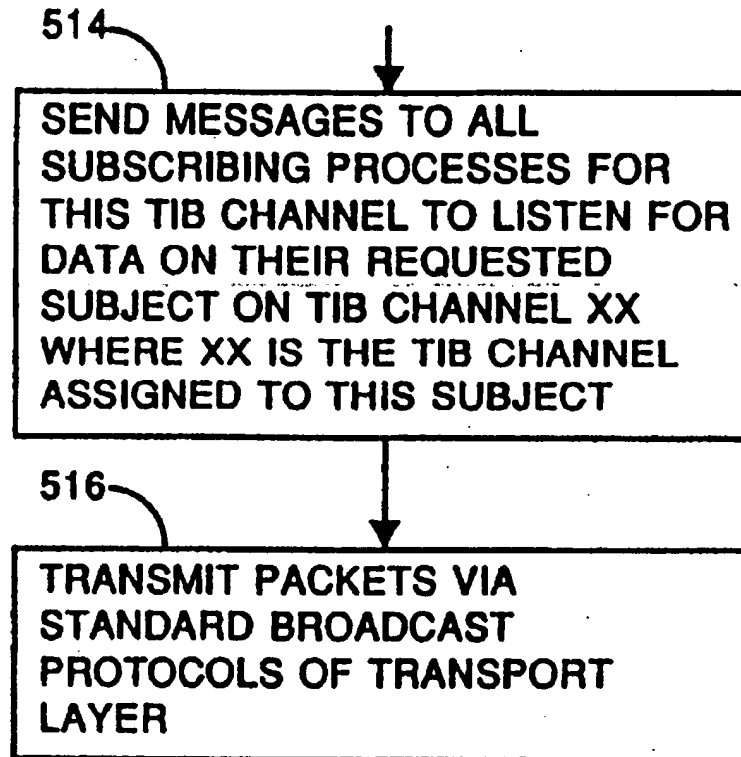


FIGURE 21 B

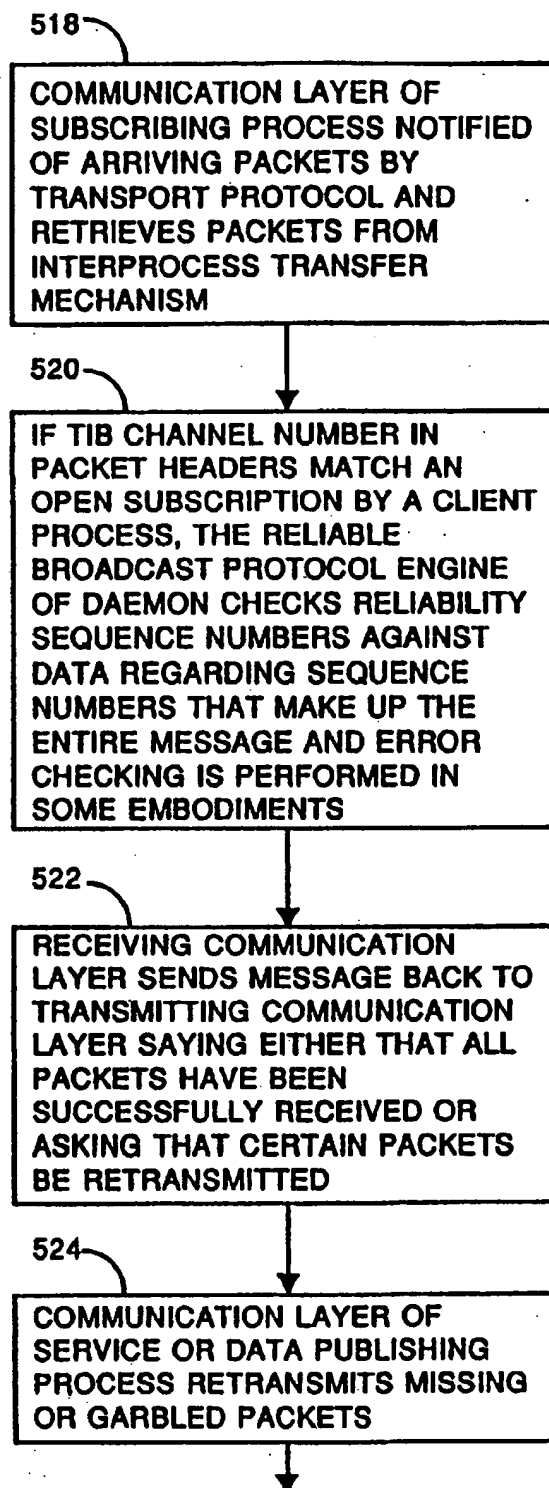


FIGURE 22 A

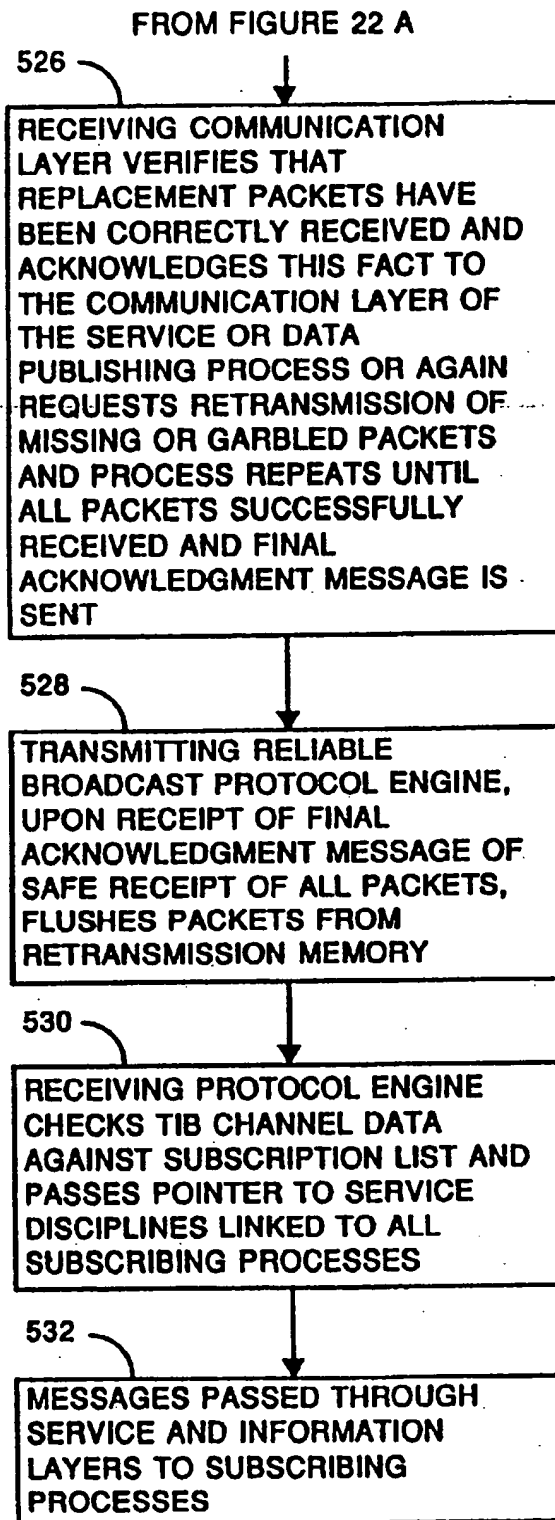


FIGURE 22 B

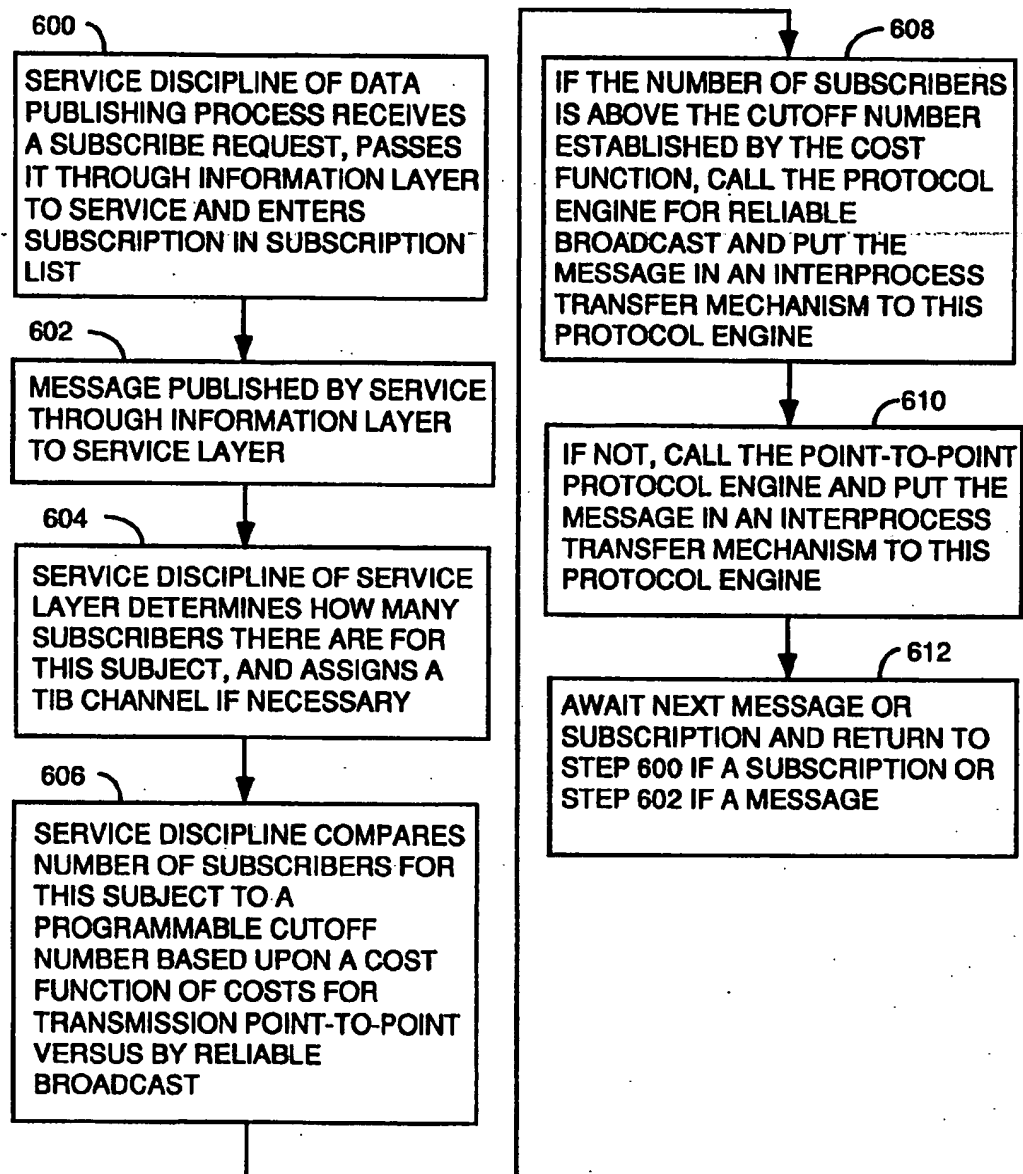


FIGURE 23

APPARATUS AND METHOD FOR PROVIDING DECOUPLING OF DATA EXCHANGE DETAILS FOR PROVIDING HIGH PERFORMANCE COMMUNICATION BETWEEN SOFTWARE PROCESSES

This is a continuation-in-part application of a U.S. patent application entitled "Apparatus and Method for Providing Decoupling of Data Exchange Details for Providing High Performance Communications Between Software Processes", U.S. Ser. No. 07/601,117, now U.S. Pat. No. 5,257,369, filed Oct. 22, 1990 which is a continuation-in-part application of U.S. Ser. No. 07/386,584, now U.S. Pat. No. 5,187,787 filed Jul. 27, 1989.

BACKGROUND OF THE INVENTION

The invention pertains to the field of decoupled information exchange between software processes running on different or even the same computer where the software processes may use different formats for data representation and organization or may use the same formats and organization but said formats and organization may later be changed without requiring any reprogramming. Also, the software processes use "semantic" or field-name information in such a way that each process can understand and use data it has received from any foreign software process, regardless of semantic or field name differences. The semantic information is decoupled from data representation and organization information.

With the proliferation of different types of computers and software programs and the ever-present need for different types of computers running different types of software programs to exchange data, there has arisen a need for a system by which such exchanges of data can occur. Typically, data that must be exchanged between software modules that are foreign to each other comprises text, data and graphics. However, there occasionally arises the need to exchange digitized voice or digitized image data or other more exotic forms of information. These different types of data are called "primitives." A software program can manipulate only the primitives that it is programmed to understand and manipulate. Other types of primitives, when introduced as data into a software program, will cause errors.

"Foreign," as the term is used herein, means that the software modules or host computers involved in the exchange "speak different languages." For example, the Motorola and Intel microprocessor widely used in personal computers and work stations use different data representations in that in one family of microprocessors the most significant byte of multibyte words is placed first while in the other family of processors the most significant byte is placed last. Further, in IBM computers text letters are coded in EBCDIC code while in almost all other computers text letters are coded in ASCII code. Also, there are several different ways of representing numbers including integer, floating point, etc. Further, foreign software modules use different ways of organizing data and use different semantic information, i.e., what each field in a data record is named and what it means.

The use of various formats for data representation and organization means that translations either to a common language or from the language of one computer or process to the language of another computer or process must be made before meaningful communication can take place.

Further, many software modules between which communication is to take place reside on different computers that are physically distant from each other and connected only local area networks, wide area networks, gateways, satellites, etc. These various networks have their own widely diverse protocols for communication. Also, at least in the world of financial services, the various sources of raw data such as Dow Jones News or Telerate™ use different data formats and communication protocols which must be understood and followed to receive data from these sources.

In complex data situations such as financial data regarding equities, bonds, money markets, etc., it is often useful to have nesting of data. That is, data regarding a particular subject is often organized as a data record having multiple "fields," each field pertaining to a different aspect of the subject. It is often useful to allow a particular field to have subfields and a particular subfield to have its own subfields and so on for as many levels as necessary. For purposes of discussion herein, this type of data organization is called "nesting." The names of the fields and what they mean relative to the subject will be called the "semantic information" for purposes of discussion herein. The actual data representation for a particular field, i.e., floating point, integer, alphanumeric, etc., and the organization of the data record in terms of how many fields it has, which are primitive fields which contain only data, and which are nested fields which contain subfields, is called the "format" or "type" information for purposes of discussion herein. A field which contains only data (and has no nested subfields) will be called a "primitive field," and a field which contains other fields will be called a "constructed field" herein.

There are two basic types of operations that can occur in exchanges of data between software modules. The first type of operation is called a "format operation" and involves conversion of the format of one data record (hereafter data records may sometimes be called "a forms") to another format. An example of such a format operation might be conversion of data records with floating point and EBCDIC fields to data records having the packed representation needed for transmission over an ETHERNET™ local area network. At the receiving process end another format operation for conversion from the ETHERNET™ packet format to integer and ASCII fields at the receiving process or software module might occur. Another type of operation will be called herein a "semantic-dependent operation" because it requires access to the semantic information as well as to the type or format information about a form to do some work on the form such as to supply a particular field of that form, e.g., today's IBM stock price or yesterday's IBM low price, to some software module that is requesting same.

Still further, in today's environment, there are often multiple sources of different types of data and/or multiple sources of the same type of data where the sources overlap in coverage but use different formats and different communication protocols (or even overlap with the same format and the same communication protocol). It is useful for a software module (software modules may hereafter be sometimes referred to as "applications") to be able to obtain information regarding a particular subject without knowing the network address of the service that provides information of that type and without knowing the details of the particular communication protocol needed to communicate with that information source.

A need has arisen therefore for a communication system which can provide an interface between diverse software modules, processes and computers for reliable, meaningful exchanges of data while "decoupling" these software mod-

ules and computers. "Decoupling" means that the software module programmer can access information from other computers or software processes without knowing where the other software modules and computers are in a network, the format that forms and data take on the foreign software, what communication protocols are necessary to communicate with the foreign software modules or computers, or what communication protocols are used to transit any networks between the source process and the destination process; and without knowing which of a multiple of sources of raw data can supply the requested data. Further, "decoupling," as the term is used herein, means that data can be requested at one time and supplied at another and that one process may obtain desired data from the instances of forms created with foreign format and foreign semantic data through the exercise by a communication interface of appropriate semantic operations to extract the requested data from the foreign forms with the extraction process being transparent to the requesting process.

Various systems exist in the prior art to allow information exchange between foreign software modules with various degrees of decoupling. One such type of system is any electronic mail software which implements Electronic Document Exchange Standards including CCITT's X.409 standard. Electronic mail software decouples applications in the sense that format or type data is included within each instance of a data record or form. However, there are no provisions for recording or processing of semantic information. Semantic operations such as extraction or translation of data based upon the name or meaning of the desired field in the foreign data structure is therefore impossible. Semantic-Dependent Operations are very important if successful communication is to occur. Further, there is no provision in Electronic Mail Software by which subject-based addressing can be implemented wherein the requesting application simply asks for information by subject without knowing the address of the source of information of that type. Further, such software cannot access a service or network for which a communication protocol has not already been established.

Relational Database Software and Data Dictionaries are another example of software systems in the prior art for allowing foreign processes to share data. The shortcoming of this class of software is that such programs can handle only "flat" tables, records and fields within records but not nested records within records. Further, the above-noted shortcoming in Electronic Mail Software also exists in Relational Database Software.

SUMMARY OF THE INVENTION

According to the teachings of the invention, there is provided a method and apparatus for providing a structure to interface foreign processes and computers while providing a degree of decoupling heretofore unknown.

The data communication interface software system according to the teachings of the invention consists essentially of several libraries of programs organized into two major components, a communication component and a data-exchange component. Interface, as the term is used herein the context of the invention, means a collection of functions which may be invoked by the application to do useful work in communicating with a foreign process or a foreign computer or both. Invoking functions of the interface may be by subroutine calls from the application or from another component in the communications interface according to the invention.

In the preferred embodiment, the functions of the interface are carried out by the various subroutines in the libraries of subroutines which together comprise the interface. Of course, those skilled in the art will appreciate that separate programs or modules may be used instead of subroutines and may actually be preferable in some cases.

Data format decoupling is provided such that a first process using data records or forms having a first format can communicate with a second process which has data records having a second, different format without the need for the first process to know or be able to deal with the format used by the second process. This form of decoupling is implemented via the data-exchange component of the communication interface software system.

The data-exchange component of the communication interface according to the teachings of the invention includes a forms-manager module and a forms-class manager module. The forms-manager module handles the creation, storage, recall and destruction of instances of forms and calls to the various functions of the forms-class manager. The latter handles the creation, storage, recall, interpretation, and destruction of forms-class descriptors which are data records which record the format and semantic information that pertain to particular classes of forms. The forms-class manager can also receive requests from the application or another component of the communication interface to get a particular field of an instance of a form when identified by the name or meaning of the field, retrieve the appropriate form instance, and deliver the requested data in the appropriate field. The forms-class manager can also locate the class definition of an unknown class of forms by looking in a known repository of such class definitions or by requesting the class definition from the forms-class manager linked to the foreign process which created the new class of form. Semantic data, such as field names, is decoupled from data representation and organization in the sense that semantic information contains no information regarding data representation or organization. The communication interface of the invention implements data decoupling in the semantic sense and in the data format sense. In the semantic sense, decoupling is implemented by virtue of the ability to carry out semantic-dependent operations. These operations allow any process coupled to the communications interface to exchange data with any other process which has data organized either the same or in a different manner by using the same field names for data which means the same thing in the preferred embodiment. In an alternative embodiment semantic-dependent operations implement an aliasing or synonym conversion facility whereby incoming data fields having different names but which mean a certain thing are either relabeled with field names understood by the requesting process or are used as if they had been so relabeled.

The interface according to the teachings of the invention has a process architecture organized in 3 layers.

Architectural decoupling is provided by an information layer such that a requesting process can request data regarding a particular subject without knowing the network address of the server or process where the data may be found. This form of decoupling is provided by a subject-based addressing system within the information layer of the communication component of the interface.

Subject-based addressing is implemented by the communication component of the communication interface of the invention by subject mapping. The communication component receives "subscribe" requests from an application which specifies the subject upon which data is requested. A

subject-mapper module in the information layer receives the request from the application and then looks up the subject in a database, table or the like. The database stores "service records" which indicate the various server processes that supply data on various subjects. The appropriate service record identifying the particular server process that can supply data of the requested type and the communication protocol (hereafter sometimes called the service discipline) to use in communicating with the identified server process is returned to the subject-mapper module.

The subject mapper has access to a plurality of communications library programs or subroutines on the second layer of the process architecture called the service layer. The routines on the service layer are called "service disciplines." Each service discipline encapsulates a predefined communication protocol which is specific to a server process. The subject mapper then invokes the appropriate service discipline identified in the service record.

The service discipline is given the subject by the subject mapper and proceeds to establish communications with the appropriate server process. Thereafter, instances of forms containing data regarding the subject are sent by the server process to the requesting process via the service discipline which established the communication.

Service protocol decoupling is provided by the service layer.

Temporal decoupling is implemented in some service disciplines directed to page-oriented server processes such as Telerate™ by access to real-time data bases which store updates to pages to which subscriptions are outstanding.

A third layer of the distributed communication component is called the communication layer and provides configuration decoupling. This layer includes a DCC library of programs that receives requests to establish data links to a particular server and determines the best communication protocol to use for the link unless the protocol is already established by the request. The communication layer also includes protocol engines to encapsulate various communication protocols such as point-to-point, broadcast, reliable broadcast and the Intelligent Multicast™ protocol. Some of the functionality of the communication layer augments the functionality of the standard transport protocols of the operating system and provides value added services.

One of these value added services is the reliable broadcast protocol. This protocol engine adds sequence numbers to packets of packetized messages on the transmit side and verifies that all packets have been received on the receive side. Packets are stored for retransmission on the transmit side. On the receive side, if all packets did not come in or some are garbled, a request is sent for retransmission. The bad or missing packets are then resent. When all packets have been successfully received, an acknowledgment message is sent. This causes the transmit side protocol engine to flush the packets out of the retransmit buffer to make room for packets of the next message.

Another value added service is the Intelligent Multicast Protocol. This protocol involves the service discipline examining the subject of a message to be sent and determining how many subscribers there are for this message subject. If the number of subscribers is below a threshold set by determining costs of point-to-point versus broadcast transmission, the message is sent point-to-point. Otherwise the message is sent by the reliable broadcast protocol.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram illustrating the relationships of the various software modules of the communication inter-

face of one embodiment of the invention to client applications and the network.

FIG. 2 is an example of a form-class definition of the constructed variety.

FIG. 3 is an example of another constructed form-class definition.

FIG. 4 is an example of a constructed form-class definition containing fields that are themselves constructed forms. Hence, this is an example of nesting.

FIG. 5 is an example of three primitive form classes.

FIG. 6 is an example of a typical form instance as it is stored in memory.

FIG. 7 illustrates the partitioning of semantic data, format data, and actual or value data between the form-class definition and the form instance.

FIG. 8 is a flow chart of processing during a format operation.

FIG. 9 is a target format-specific table for use in format operations.

FIG. 10 is another target format-specific table for use in format operations.

FIG. 11 is an example of a general conversion table for use in format operations.

FIG. 12 is a flow chart for a typical semantic-dependent operation.

FIGS. 13A and 13B are, respectively, a class definition and the class descriptor form which stores this class definition.

FIG. 14 is a block diagram illustrating the relationships between the subject-mapper module and the service discipline modules of the communication component to the requesting application and the service for subject-based addressing.

FIG. 15 illustrates the relationship of the various modules, libraries and interfaces of an alternative embodiment of the invention to the client applications.

FIG. 16 illustrates the relationships of various modules inside the communication interface of an alternative embodiment.

FIG. 17 is a block diagram of a typical distributed computer network.

FIG. 18 is a process architecture showing the relationship of the DCC library to the DCC protocol engines in the daemon.

FIG. 19, comprised of FIGS. 19A and 19B, is a flow diagram of the process which occurs, inter alia, at the three layers of the software of the invention where a subscribe request is sent to a service.

FIG. 20, comprised of FIGS. 20A and 20B, is a flow chart of the process which occurs at, inter alia, the three layers of the software interface according to the teachings of the invention when a subscribe request is received at a data producing process and messages flow back to the subscribing process.

FIG. 21, comprised of FIGS. 21A and 21B, is a flow chart of the process which occurs at the DCC library and in the reliable broadcast protocol engine when messages are sent by the reliable broadcast protocol.

FIG. 22, comprised of FIGS. 22A and 22B, is a flow chart of processing by a reliable broadcast protocol engine on the data consumer side of the reliable broadcast transaction.

FIG. 23 is a flow chart of the processing which occurs in the service discipline to implement the Intelligent Multicast™ protocol.

DETAILED DESCRIPTION OF THE PREFERRED AND ALTERNATIVE EMBODIMENTS

Since the following description is highly technical, it can best be understood by an understanding of the terms used in the digital network telecommunication art defined in the appended glossary. The reader is urged to read the glossary at the end of the specification herein first.

Referring to FIG. 1 there is shown a block diagram of a typical system in which the communications interface of the invention could be incorporated, although a wide variety of system architectures can benefit from the teachings of the invention. The communication interface of the invention may be sometimes hereafter referred to as the TIB™ or Teknekron Information Bus in the specification of an alternative embodiment given below. The reader is urged at this point to study the glossary of terms included in this specification to obtain a basic understanding of some of the more important terms used herein to describe the invention. The teachings of the invention are incorporated in several libraries of computer programs which, taken together, provide a communication interface having many functional capabilities which facilitate modularity in client application development and changes in network communication or service communication protocols by coupling of various client applications together in a "decoupled" fashion. Hereafter, the teachings of the invention will be referred to as the communication interface. "Decoupling," as the term is used herein, means that the programmer of client application is freed of the necessity to know the details of the communication protocols, data representation format and data record organization of all the other applications or services with which data exchanges are desired. Further, the programmer of the client application need not know the location of services or servers providing data on particular subjects in order to be able to obtain data on these subjects. The communication interface automatically takes care of all the details in data exchanges between client applications and between data-consumer applications and data-provider services.

The system shown in FIG. 1 is a typical network coupling multiple host computers via a network or by shared memory. Two host computers, 10 and 12, are shown in FIG. 1 running two client applications 16 and 18, although in other embodiments these two client applications may be running on the same computer. These host computers are coupled by a network 14 which may be any of the known networks such as the ETHERNET™ communication protocol, the token ring protocol, etc. A network for exchanging data is not required to practice the invention, as any method of exchanging data known in the prior art will suffice for purposes of practicing the invention. Accordingly, shared memory files or shared distributed storage to which the host computers 10 and 12 have equal access will also suffice as the environment in which the teachings of the invention are applicable.

Each of the host computers 10 and 12 has random access memory and bulk memory such as disk or tape drives associated therewith (not shown). Stored in these memories are the various operating system programs, client application programs, and other programs such as the programs in the libraries that together comprise the communication interface which cause the host computers to perform useful work. The libraries of programs in the communication interface provide basic tools which may be called upon by client applications to do such things as find the location of services

that provide data on a particular subject and establish communications with that service using the appropriate communication protocol.

Each of the host computers may also be coupled to user interface devices such as terminals, printers, etc. (not shown).

In the exemplary system shown in FIG. 1, host computer 10 has stored in its memory a client application program 16. Assume that this client application program 16 requires exchanges of data with another client application program or service 18 controlling host computer 12 in order to do useful work. Assume also that the host computers 10 and 12 use different formats for representation of data and that application programs 16 and 18 also use different formats for data representation and organization for the data records created thereby. These data records will usually be referred to herein as forms. Assume also that the data path 14 between the host computers 10 and 12 is comprised of a local area network of the ETHERNET™ variety.

Each of the host processors 10 and 12 is also programmed with a library of programs, which together comprise the communication interfaces 20 and 22, respectively. The communication interface programs are either linked to the compiled code of the client applications by a linker to generate run time code, or the source code of the communication programs is included with the source code of the client application programs prior to compiling. In any event, the communication library programs are somehow bound to the client application. Thus, if host computer 10 was running two client applications, each client application would be bound to a communication interface module such as module 20.

The purpose of the communications interface module 20 is to decouple application 16 from the details of the data format and organization of data in forms used by application 18, the network address of application 18, and the details of the communication protocol used by application 18, as well as the details of the data format and organization and communication protocol necessary to send data across network 14. Communication interface module 22 serves the same function for application 18, thereby freeing it from the need to know many details about the application 16 and the network 14. The communication interface modules facilitate modularity in that changes can be made in client applications, data formats or organizations, host computers, or the networks used to couple all of the above together without the need for these changes to ripple throughout the system to ensure continued compatibility.

In order to implement some of these functions, the communications interfaces 20 and 22 have access via the network 14 to a network file system 24 which includes a subject table 26 and a service table 28. These tables will be discussed in more detail below with reference to the discussion of subject-based addressing. These tables list the network addresses of services that provide information on various subjects.

A typical system model in which the communication interface is used consists of users, users groups, networks, services, service instances (or servers) and subjects. Users, representing human end users, are identified by a user-ID. The user ID used in the communications interface is normally the same as the user ID or log-on ID used by the underlying operating system (not shown). However, this need not be the case. Each user is a member of exactly one group.

Groups are comprised of users with similar service access patterns and access rights. Access rights to a service or

system object are grantable at the level of users and at the level of groups. The system administrator is responsible for assigning users to groups.

A "network," as the term is used herein, means the underlying "transport layer" (as the term is used in the ISO network layer model) and all layers beneath the transport layer in the ISO network model. An application can send or receive data across any of the networks to which its host computer is attached.

The communication interface according to the teachings of the invention, of which blocks 20 and 22 in FIG. 1 are exemplary, includes for each client application to which it is bound a communications component 30 and a data-exchange component 32. The communications component 30 is a common set of communication facilities which implement, for example, subject-based addressing and/or service discipline decoupling. The communications component is linked to each client application. In addition, each communications component is linked to the standard transport layer protocols, e.g., TCP/IP, of the network to which it is coupled. Each communication component is linked to and can support multiple transport layer protocols. The transport layer of a network does the following things: it maps transport layer addresses to network addresses, multiplexes transport layer connections onto network connections to provide greater throughput, does error detection and monitoring of service quality, error recovery, segmentation and blocking, flow control of individual connections of transport layer to network and session layers, and expedited data transfer. The communications component cooperates with the transport layer to provide reliable communications protocols for client applications as well as providing location transparency and network independence to the client applications.

The data-exchange component of the communications interface, of which component 32 is typical, implements a powerful way of representing and transmitting data by encapsulating the data within self-describing data objects called forms. These forms are self-describing in that they include not only the data of interest, but also type or format information which describes the representations used for the data and the organization of the form. Because the forms include this type or format information, format operations to convert a particular form having one format to another format can be done using strictly the data in the form itself without the need for access to other data called class descriptors or class definitions which give semantic information. Semantic information in class descriptors basically means the names of the fields of the form.

The ability to perform format operations solely with the data in the form itself is very important in that it prevents the delays encountered when access must be made to other data objects located elsewhere, such as class descriptors. Since format operations alone typically account for 25 to 50% of the processing time for client applications, the use of self-describing objects streamlines processing by rendering it faster.

The self-describing forms managed by the data-exchange component also allow the implementation of generic tools for data manipulation and display. Such tools include communication tools for sending forms between processes in a machine-independent format. Further, since self-describing forms can be extended, i.e., their organization changed or expanded, without adversely impacting the client applications using said forms, such forms greatly facilitate modular application development.

Since the lowest layer of the communications interface is linked with the transport layer of the ISO model and since

the communications component 30 includes multiple service disciplines and multiple transport-layer protocols to support multiple networks, it is possible to write application-oriented protocols which transparently switch over from one network to another in the event of a network failure.

A "service" represents a meaningful set of functions which are exported by an application for use by its client applications. Examples of services are historical news retrieval services such as Dow Jones New, Quotron data feed, and a trade ticket router. Applications typically export only one service, although the export of many different services is also possible.

A "service instance" is an application or process capable of providing the given service. For a given service, several "instances" may be concurrently providing the service so as to improve the throughput of the service or provide fault tolerance.

Although networks, services and servers are traditional components known in the prior art, prior art distributed systems do not recognize the notion of a subject space or data independence by self-describing, nested data objects. Subject space supports one form of decoupling called subject-based addressing. Self-describing data objects which may be nested at multiple levels are new. Decoupling of client applications from the various communications protocols and data formats prevalent in other parts of the network is also very useful.

The subject space used to implement subject-based addressing consists of a hierarchical set of subject categories. In the preferred embodiment, a four-level subject space hierarchy is used. An example of a typical subject is: "equity.ibm.composite.trade." The client applications coupled to the communications interface have the freedom and responsibility to establish conventions regarding use and interpretations of various subject categories.

Each subject is typically associated with one or more services providing data about that subject in data records stored in the system files. Since each service will have associated with it in the communication components of the communication interface a service discipline, i.e., the communication protocol or procedure necessary to communicate with that service, the client applications may request data regarding a particular subject without knowing where the service instances that supply data on that subject are located on the network by making subscription requests giving only the subject without the network address of the service providing information on that subject. These subscription requests are translated by the communications interface into an actual communication connection with one or more service instances which provide information on that subject.

A set of subject categories is referred to as a subject domain. Multiple subject domains are allowed. Each domain can define domain-specific subject and coding functions for efficiently representing subjects in message headers.

DATA INDEPENDENCE: The Data-Exchange Component

The overall purpose of the data-exchange component such as component 32 in FIG. 1 of the communication interface is to decouple the client applications such as application 16 from the details of data representation, data structuring and data semantics.

Referring to FIG. 2, there is shown an example of a class definition for a constructed class which defines both format and semantic information which is common to all instances

11

of forms of this class. In the particular example chosen, the form class is named `Player_Name` and has a class ID of 1000. The instances of forms of this class 1000 include data regarding the names, ages and NTRP ratings for tennis players. Every class definition has associated with it a class number called the class ID which uniquely identifies the class.

The class definition gives a list of fields by name and the data representation of the contents of the field. Each field contains a form and each form may be either primitive or constructed. Primitive class forms store actual data, while constructed class forms have fields which contain other forms which may be either primitive or constructed. In the class definition of FIG. 2, there are four fields named `Rating`, `Age`, `Last_Name` and `First_Name`. Each field contains a primitive class form so each field in instances of forms of this class will contain actual data. For example, the field `Rating` will always contain a primitive form of class 11. Class 11 is a primitive class named `Floating_Point` which specifies a floating-point data representation for the contents of this field. The primitive class definition for the class `Floating_Point`, class 11, is found in FIG. 5. The class definition of the primitive class 11 contains the class name, `Floating_Point`, which uniquely identifies the class (the class number, class 11 in this example, also uniquely identifies the class) and a specification of the data representation of the single data value. The specification of the single data value uses well-known predefined system data types which are understood by both the host computer and the application dealing with this class of forms.

Typical specifications for data representation of actual data values include integer, floating point, ASCII character strings or EBCDIC character strings, etc. In the case of primitive class 11, the specification of the data value is `Floating_Point_1/1` which is an arbitrary notation indicating that the data stored in instances of forms of this primitive class will be floating-point data having two digits total, one of which is to the right of the decimal point.

Returning to the consideration of the `Player_Name` class definition of FIG. 2, the second field is named `Age`. This field contains forms of the primitive class named `Integer` associated with class number 12 and defined in FIG. 5. The `Integer` class of form, class 12, has, per the class definition of FIG. 5, a data representation specification of `Integer_3`, meaning the field contains integer data having three digits. The last two fields of the class 1000 definition in FIG. 2 are `Last_Name` and `First_Name`. Both of these fields contain primitive forms of a class named `String_Twenty_ASCII`, class 10. The class 10 class definition is given in FIG. 5 and specifies that instances of forms of this class contain ASCII character strings which are 20 characters long.

FIG. 3 gives another constructed class definition named `Player_Address`, class 1001. Instances of forms of this class each contain three fields named `Street`, `City` and `State`. Each of these three fields contains primitive forms of the class named `String_20_ASCII`, class 10. Again, the class definition for class 10 is given in FIG. 5 and specifies a data representation of 20-character ASCII strings.

An example of the nesting of constructed class forms is given in FIG. 4. FIG. 4 is a class definition for instances of forms in the class named `Tournament_Entry`, class 1002. Each instance of a form in this class contains three fields named `Tournament_Name`, `Player`, and `Address`. The field `Tournament_Name` includes forms of the primitive class named `String_Twenty_ASCII`, class 10 defined in FIG. 5. The field named `Player` contains instances of constructed

12

forms of the class named `Player_Name`, class 1000 having the format and semantic characteristics given in FIG. 2. The field named `Address` contains instances of the constructed form of constructed forms of the constructed class named `Player_Address`, class 1001, which has the format and semantic characteristics given in the class definition of FIG. 3.

The class definition of FIG. 4 shows how nesting of forms can occur in that each field of a form is a form itself and every form may be either primitive and have only one field or constructed and have several fields. In other words, instances of a form may have as many fields as necessary, and each field may have as many subfields as necessary. Further, each subfield may have as many sub-subfields as necessary. This nesting goes on for any arbitrary number of levels. This data structure allows data of arbitrary complexity to be easily represented and manipulated.

Referring to FIG. 6 there is shown an instance of a form of the class of forms named `Tournament_Entry`, class 1002, as stored as an object in memory. The block of data 38 contains the constructed class number 1002 indicating that this is an instance of a form of the constructed class named `Tournament_Entry`. The block of data 40 indicates that this class of form has three fields. Those three fields have blocks of data shown at 42, 44, and 46 containing the class numbers of the forms in these fields. The block of data at 42 indicates that the first field contains a form of class 10 as shown in FIG. 5. A class 10 form is a primitive form containing a 20-character string of ASCII characters as defined in the class definition for class 10 in FIG. 5. The actual string of ASCII characters for this particular instance of this form is shown at 48, indicating that this is a tournament entry for the U.S. Open tennis tournament. The block of data at 44 indicates that the second field contains a form which is an instance of a constructed form of class 1000. Reference to this class definition shows that this class is named `Player_Name`. The block of data 50 shows that this class of constructed form contains four subfields. Those fields contain forms of the classes recorded in the blocks of data shown at 52, 54, 56 and 58. These fields would be subfields of the field 44. The first subfield has a block of data at 52, indicating that this subfield contains a form of primitive class 11. This class of form is defined in FIG. 5 as containing a floating-point two-digit number with one decimal place. The actual data for this instance of the form is shown at 60, indicating that this player has an NTRP rating of 3.5. The second subfield has a block of data at 54, indicating that this subfield contains a form of primitive class 12. The class definition for this class indicates that the class is named `integer` and contains integer data. The class definition for class 1000 shown in FIG. 2 indicates that this integer data, shown at block 62, is the player's age. Note that the class definition semantic data regarding field names is not stored in the form instance. Only the format or type information is stored in the form instance in the form of the class ID for each field.

The third subfield has a block of data at 56, indicating that this subfield contains a form of primitive class 10 named `String_20_ASCII`. This subfield corresponds to the field `Last_Name` in the form of class `Player_Name`, class 1000, shown in FIG. 2. The primitive class 10 class definition specifies that instances of this primitive class contain a 20-character ASCII string. This string happens to define the player's last name. In the instance shown in FIG. 6, the player's last name is `Blackett`, as shown at 64.

The last subfield has a block of data at 58, indicating that the field contains a primitive form of primitive class 10

which is a 20-character ASCII string. This subfield is defined in the class definition of class 1000 as containing the player's first name. This ASCII string is shown at 66.

The third field in the instance of the form of class 1002 has a block of data at 46, indicating that this field contains a constructed form of the constructed class 1001. The class definition for this class is given in FIG. 3 and indicates the class is named Player_Address. The block of data at 68 indicates that this field has three subfields containing forms of the class numbers indicated at 70, 72 and 74. These subfields each contain forms of the primitive class 10 defined in FIG. 5. Each of these subfields therefore contains a 20-character ASCII string. The contents of these three fields are defined in the class definition for class 1001 and are, respectively, the street, city and state entries for the address of the player named in the field 44. These 3-character strings are shown at 76, 78 and 80, respectively.

Referring to FIG. 7, there is shown a partition of the semantic information, format information and actual data between the class definition and instances of forms of this class. The field name and format or type information are stored in the class definition, as indicated by box 82. The format or type information (in the form of the class ID) and actual data or field values are stored in the instance of the form as shown by box 84. For example, in the instance of the form of class Tournament_Entry, class 1002 shown in FIG. 6, the format data for the first field is the data stored in block 42, while the actual data for the first field is the data shown at block 48. Essentially, the class number or class ID is equated by the communications interface with the specification for the type of data in instances of forms of that primitive class. Thus, the communications interface can perform format operations on instances of a particular form using only the format data stored in the instance of the form itself without the need for access to the class definition. This speeds up format operations by eliminating the need for the performance of the steps required to access a class definition which may include network access and/or disk access, which would substantially slow down the operation. Since format-type operations comprise the bulk of all operations in exchanging data between foreign processes, the data structure and the library of programs to handle the data structure defined herein greatly increase the efficiency of data exchange between foreign processes and foreign computers.

For example, suppose that the instance of the form shown in FIG. 6 has been generated by a process running on a computer by Digital Equipment Corporation (DEC) and therefore text is expressed in ASCII characters. Suppose also that this form is to be sent to a process running on an IBM computer, where character strings are expressed in EBCDIC code. Suppose also that these two computers were coupled by a local area network using the ETHERNET™ communications protocol.

To make this transfer, several format operations would have to be performed. These format operations can best be understood by reference to FIG. 1 with the assumption that the DEC computer is host 1 shown at 10 and the IBM computer is host 2 shown at 12.

The first format operation to transfer the instance of the form shown in FIG. 6 from application 16 to application 18 would be a conversion from the format shown in FIG. 6 to a packed format suitable for transfer via network 14. Networks typically operate on messages comprised of blocks of data comprising a plurality of bytes packed together end to end preceded by multiple bytes of header information which include such things as the message length, the destination

address, the source address, and so on, and having error correction code bits appended to the end of the message. Sometimes delimiters are used to mark the start and end of the actual data block.

The second format operation which would have to be performed in this hypothetical transfer would be a conversion from the packed format necessary for transfer over network 14 to the format used by the application 18 and the host computer 12.

Format operations are performed by the forms-manager modules of the communications interface. For example, the first format operation in the hypothetical transfer would be performed by the forms-manager module 86 in FIG. 1, while the second format operation in the hypothetical transfer would be performed by the forms-manager module in the data-exchange component 88.

Referring to FIG. 8, there is shown a flowchart of the operations performed by the forms-manager modules in performing format operations. Further details regarding the various functional capabilities of the routines in the forms-manager modules of the communications interface will be found in the functional specifications for the various library routines of the communications interface included herein. The process of FIG. 8 is implemented by the software programs in the forms-manager modules of the data-exchange components in the communications interface according to the teachings of the invention. The first step is to receive a format conversion call from either the application or from another module in the communications interface. This process is symbolized by block 90 and the pathways 92 and 94 in FIG. 1. The same type call can be made by the application 18 or the communications component 96 for the host computer 12 in FIG. 1 to the forms-manager module in the data-exchange component 88, since this is a standard functional capability or "tool" provided by the communication interface of the invention to all client applications. Every client application will be linked to a communication interface like interface 20 in FIG. 1.

Typically, format conversion calls from the communication components such as modules 30 and 96 in FIG. 1 to the forms-manager module will be from a service discipline module which is charged with the task of sending a form in format 1 to a foreign application which uses format 2. Another likely scenario for a format conversion call from another module in the communication interface is when a service discipline has received a form from another application or service which is in a foreign format and which needs to be converted to the format of the client application.

The format conversion call will have parameters associated with it which are given to the forms manager. These parameters specify both the "from" format and the "to" or "target" format.

Block 98 represents the process of accessing an appropriate target format-specific table for the specified conversion, i.e., the specified "from" format and the specified "to" format will have a dedicated table that gives details regarding the appropriate target format class for each primitive "from" format class to accomplish the conversion. There are two tables which are accessed sequentially during every format conversion operation in the preferred embodiment. In alternative embodiments, these two tables may be combined. Examples of the two tables used in the preferred embodiment are shown in FIGS. 9, 10 and 11. FIG. 9 shows a specific format conversion table for converting from DEC machines to X.409 format. FIG. 10 shows a format-specific conversion table for converting from X.409 format to IBM

15

machine format. FIG. 11 shows a general conversion procedures table identifying the name of the conversion program in the communications interface library which performs the particular conversion for each "from"-"to" format pair.

The tables of FIGS. 9 and 10 probably would not be the only tables necessary for sending a form from the application 16 to the application 18 in FIG. 1. There may be further format-specific tables necessary for conversion from application 16 format to DEC machine format and for conversion from IBM machine format to application 18 format. However, the general concept of the format conversion process implemented by the forms-manager modules of the communications interface can be explained with reference to FIGS. 9, 10 and 11.

Assume that the first conversion necessary in the process of sending a form from application 16 to application 18 is a conversion from DEC machine format to a packed format suitable for transmission over an ETHERNET™ network. In this case, the format conversion call received in step 90 would invoke processing by a software routine in the forms-manager module which would perform the process symbolized by block 98.

In this hypothetical example, the appropriate format-specific table to access by this routine would be determined by the "from" format and "to" format parameters in the original format conversion call received by block 90. This would cause access to the table shown in FIG. 9. The format conversion call would also identify the address of the form to be converted.

The next step is symbolized by block 100. This step involves accessing the form identified in the original format conversion call and searching through the form to find the first field containing a primitive class of form. In other words, the record is searched until a field is found storing actual data as opposed to another constructed form having subfields.

In the case of the form shown in FIG. 6, the first field storing a primitive class of form is field 42. The "from" column of the table of FIG. 9 would be searched using the class number 10 until the appropriate entry was found. In this case, the entry for a "from" class of 10 indicates that the format specified in the class definition for primitive class 25 is the "to" format. This process of looking up the "to" format using the "from" format is symbolized by block 102 in FIG. 8. The table shown in FIG. 9 may be "hardwired" into the code of the routine which performs the step symbolized by block 102.

Alternatively, the table of FIG. 9 may be a database or other file stored somewhere in the network file system 24 in FIG. 1. In such a case, the routine performing the step 102 in FIG. 8 would know the network address and file name for the file to access for access to the table of FIG. 9.

Next, the process symbolized by block 104 in FIG. 8 is performed by accessing the general conversion procedures table shown in FIG. 11. This is a table which identifies the conversion program in the forms manager which performs the actual work of converting one primitive class of form to another primitive class of form. This table is organized with a single entry for every "from"-"to" format pair. Each entry in the table for a "from"-"to" pair includes the name of the conversion routine which does the actual work of the conversion. The process symbolized by block 104 comprises the steps of taking the "from"-"to" pair determined from access to the format-specific conversion table in step 102 and searching the entries of the general conversion proce-

16

dures table until an entry having a "from"-"to" match is found. In this case, the third entry from the top in the table of FIG. 11 matches the "from"-"to" format pair found in the access to FIG. 9. This entry is read, and it is determined that the name of the routine to perform this conversion is ASCII_ETHER. (In many embodiments, the memory address of the routine, opposed to the name, would be stored in the table.)

Block 106 in FIG. 8 symbolizes the process of calling the conversion program identified by step 104 and performing this conversion routine to change the contents of the field selected in step 100 to the "to" or target format identified in step 102. In the hypothetical example, the routine ASCII_ETHER would be called and performed by step 106. The call to this routine would deliver the actual data stored in the field selected in the process of step 100, i.e., field 42 of the instance of a form shown in FIG. 6, such that the text-string "U.S. Open" would be converted to a packed ETHERNET™ format.

Next, the test of block 108 is performed to determine if all fields containing primitive classes of forms have been processed. If they have, then format conversion of the form is completed, and the format conversion routine is exited as symbolized by block 110.

If fields containing primitive classes of forms remain to be processed, then the process symbolized by block 112 is performed. This process finds the next field containing a primitive class of form.

Thereafter, the processing steps symbolized by blocks 102, 104, 106, and 108 are performed until all fields containing primitive classes of forms have been converted to the appropriate "to" format.

As noted above, the process of searching for fields containing primitive classes of forms proceeds serially through the form to be converted. If the next field encountered contains a form of a constructed class, that class of form must itself be searched until the first field therein with a primitive class of form is located. This process continues through all levels of nesting for all fields until all fields have been processed and all data stored in the form has been converted to the appropriate format. As an example of how this works, in the form of FIG. 6, after processing the first field 42, the process symbolized by block 112 in FIG. 8 would next encounter the field 44 (fields will be referred to by the block of data that contain the class ID for the form stored in that field although the contents of the field are both the class ID and the actual data or the fields and subfields of the form stored in that field). Note that in the particular class of form represented by FIG. 6, the second field 44 contains a constructed form comprised of several subfields. Processing would then access the constructed form of class 1000 which is stored by the second field and proceeds serially through this constructed form until it locates the first field thereof which contains a form of a primitive class. In the hypothetical example of FIG. 6, the first field would be the subfield indicated by the class number 11 at 52. The process symbolized by block 102 would then look up class 11 in the "from" column in the table of FIG. 9 and determine that the target format is specified by the class definition of primitive class 15. This "from"-"to" pair 11-15 would then be compared to the entries of the table of FIG. 11 to find a matching entry. Thereafter, the process of block 106 in FIG. 8 would perform the conversion program called Float1_ETHER to convert the block of data at 60 in FIG. 6 to the appropriate ETHERNET™ packed format. The process then would continue through all levels of nesting.

Referring to FIG. 12, there is shown a flowchart for a typical semantic-dependent operation. Semantic-dependent operations allow decoupling of applications by allowing one application to get the data in a particular field of an instance of a form generated by a foreign application provided that the field name is known and the address of the form instance is known. The communications interface according to the teachings of the invention receives semantic-dependent operation requests from client applications in the form of Get_Field calls in the preferred embodiment where all processes use the same field names for data fields which mean the same thing (regardless of the organization of the form or the data representation of the field in the form generated by the foreign process). In alternative embodiments, an aliasing or synonym table or data base is used. In such embodiments, the Get_Field call is used to access the synonym table in the class manager and looks for all synonyms of the requested field name. All field names which are synonyms of the requested field name are returned. The class manager then searches the class definition for a match with either the requested field name or any of the synonyms and retrieves the field having the matching field name.

Returning to consideration of the preferred embodiment, such Get_Field calls may be made by client applications directly to the forms-class manager modules such as the module 122 in FIG. 1, or they may be made to the communications components or forms-manager modules and transferred by these modules to the forms-class manager. The forms-class manager creates, destroys, manipulates, stores and reads form-class definitions.

A Get_Field call delivers to the forms-class manager the address of the form involved and the name of the field in the form of interest. The process of receiving such a request is symbolized by block 120 in FIG. 12. Block 20 also symbolizes the process by which the class manager is given the class definition either programmatically, i.e., by the requesting application, or is told the location of a data base where the class definitions including the class definition for the form of interest may be found. There may be several databases or files in the network file system 24 of FIG. 1 wherein class definitions are stored. It is only necessary to give the forms-class manager the location of the particular file in which the class definition for the form of interest is stored.

Next, as symbolized by block 122, the class-manager module accesses the class definition for the form class identified in the original call.

The class manager then searches the class definition field names to find a match for the field name given in the original call. This process is symbolized by block 124.

After locating the field of interest in the class definition, the class manager returns a relative address pointer to the field of interest in instances of forms of this class. This process is symbolized by block 126 in FIG. 12. The relative address pointer returned by the class manager is best understood by reference to FIGS. 2, 4 and 6. Suppose that the application which made the Get_Field call was interested in determining the age of a particular player. The Get_Field request would identify the address for the instance of the form of class 1002 for player Blackett as illustrated in FIG. 6. Also included in the Get_Field request would be the name of the field of interest, i.e., "age". The class manager would then access the instance of the form of interest and read the class number identifying the particular class descriptor or class definition which applied to this class of forms. The class manager would then access the class

descriptor for class 1002 and find a class definition as shown in FIG. 4. The class manager would then access the class definitions for each of the fields of class definition 1002 and would compare the field name in the original Get_Field request to the field names in the various class definitions which make up the class definition for class 1002. In other words, the class manager would compare the names of the fields in the class definitions for classes 10, 1000, and 1001 to the field name of interest, "Age". A match would be found in the class definition for class 1000 as seen from FIG. 2. For the particular record format shown in FIG. 6, the "Age" field would be the block of data 62, which is the tenth block of data in from the start of the record. The class manager would then return a relative address pointer of 10 in block 126 of FIG. 12. This relative address pointer is returned to the client application which made the original Get_Field call. The client application then issues a Get_Data call to the forms-manager module and delivers to the forms-manager module the relative address of the desired field in the particular instance of the form of interest. The forms-manager module must also know the address of the instance of the form of interest which it will already have if the original Get_Field call came through the forms-manager module and was transferred to the forms-class manager. If the forms-manager module does not have the address of the particular instance of the form of interest, then the forms manager will request it from the client application. After receiving the Get_Data call and obtaining the relative address and the address of the instance of the form of interest, the forms manager will access this instance of the form and access the requested data and return it to the client application. This process of receiving the Get_Data call and returning the appropriate data is symbolized by block 128 in FIG. 12.

Normally, class-manager modules store the class definitions needed to do semantic-dependent operations in RAM of the host machine as class descriptors. Class definitions are the specification of the semantic and formation information that define a class. Class descriptors are memory objects which embody the class definition. Class descriptors are stored in at least two ways. In random access memory (RAM), class descriptors are stored as forms in the format native to the machine and client application that created the class definition. Class descriptors stored on disk or tape are stored as ASCII strings of text.

When the class-manager module is asked to do a semantic-dependent operation, it searches through its store of class descriptors in RAM and determines if the appropriate class descriptor is present. If it is, this class descriptor is used to perform the operation detailed above with reference to FIG. 12. If the appropriate class descriptor is not present, the class manager must obtain it. This is done by searching through known files of class descriptors stored in the system files 24 in FIG. 1 or by making a request to the foreign application that created the class definition to send the class definition to the requesting module. The locations of the files storing class descriptors are known to the client applications, and the class-manager modules also store these addresses. Often, the request for a semantic-dependent operation includes the address of the file where the appropriate class descriptor may be found. If the request does not contain such an address, the class manager looks through its own store of class descriptors and through the files identified in records stored by the class manager identifying the locations of system class descriptor files.

If the class manager asks for the class descriptor from the foreign application that generated it, the foreign application sends a request to its class manager to send the appropriate

class descriptor over the network to the requesting class manager or the requesting module. The class descriptor is then sent as any other form and used by the requesting class manager to do the requested semantic-dependent operation.

If the class manager must access a file to obtain a class descriptor, it must also convert the packed ASCII representation in which the class descriptors are stored on disk or tape to the format of a native form for storage in RAM. This is done by parsing the ASCII text to separate out the various field names and specifications of the field contents and the class numbers.

FIGS. 13A and 13B illustrate, respectively, a class definition and the structure and organization of a class descriptor for the class definition of FIG. 13A and stored in memory as a form. The class definition given in FIG. 13A is named Person_Class and has only two fields, named last and first. Each of these fields is specified to store a 20-character ASCII string.

FIG. 13B has a data block 140 which contains 1021 indicating that the form is a constructed form having a class number 1021. The data block at 142 indicates that the form has 3 fields. The first field contains a primitive class specified to contain an ASCII string which happens to store the class name, Person_Class, in data block 146. The second field is of a primitive class assigned the number 2, data block 148, which is specified to contain a boolean value, data block 150. Semantically, the second field is defined in the class definition for class 1021 to define whether the form class is primitive (true) or constructed (false). In this case, data block 150 is false indicating that class 1021 is a constructed class. The third field is a constructed class given the class number 112 as shown by data block 152. The class definition for class 1021 defines the third field as a constructed class form which gives the names and specifications of the fields in the class definition. Data block 154 indicates that two fields exist in a class 112 form. The first field of class 112 is itself a constructed class given the class number 150, data block 156, and has two subfields, data block 158. The first subfield is a primitive class 15, data block 160, which is specified in the class definition for class 150 to contain the name of the first field in class 1021. Data block 162 gives the name of the first field in class 1021. The second subfield is of primitive class 15, data block 164, and is specified in the class definition of class 150 (not shown) to contain an ASCII string which specifies the representation, data block 166, of the actual data stored in the first field of class 1021. The second field of class 112 is specified in the class definition of class 112 to contain a constructed form of class 150, data block 168, which has two fields, data block 170, which give the name of the next field in class 1021 and specify the type of representation of the actual data stored in this second field.

DATA DISTRIBUTION AND SERVICE PROTOCOL DECOUPLING BY SUBJECT-BASED ADDRESSING AND THE USE OF SERVICE DISCIPLINE PROTOCOL LAYERS

Referring to FIG. 14, there is shown a block diagram of the various software modules, files, networks, and computers which cooperate to implement two important forms of decoupling. These forms of decoupling are data distribution decoupling and service protocol decoupling. Data distribution decoupling means freeing client applications from the necessity to know the network addresses for servers providing desired services. Thus, if a particular application needs

to know information supplied by, for example, the Dow Jones news service, the client application does not need to know which servers and which locations are providing data from the Dow Jones news service raw data feed.

Service protocol decoupling means that the client applications need not know the particular communications protocols used by the servers, services or other applications with which exchanges of data are desired.

Data distribution decoupling is implemented by the communications module 30 in FIG. 14. The communications component is comprised of a library of software routines which implement a subject mapper 180 and a plurality of service disciplines to implement subject-based addressing. Service disciplines 182, 184 and 186 are exemplary of the service disciplines involved in subject-based addressing.

Subject-based addressing allows services to be modified or replaced by alternate services providing equivalent information without impacting the information consumers. This decoupling of the information consumers from information providers permits a higher degree of modularization and flexibility than that provided by traditional service-oriented models.

Subject-based addressing starts with a subscribe call 188 to the subject mapper 180 by a client application 16 running on host computer 10. The subscribe call is a request for information regarding a particular subject. Suppose hypothetically that the particular subject was equity.IBM.news. This subscribe call would pass two parameters to the subject mapper 180. One of these parameters would be the subject equity.IBM.news. The other parameter would be the name of a callback routine in the client application 16 to which data regarding the subject is to be passed. The subscribe call to the subject mapper 180 is a standard procedure call.

The purpose of the subject mapper is to determine the network address for services which provide information on various subjects and to invoke the appropriate service discipline routines to establish communications with those services. To find the location of the services which provide information regarding the subject in the subscribe call, the subject mapper 80 sends a request symbolized by line 190 to a directory-services component 192. The directory-services component is a separate process running on a computer coupled to the network 14 and in fact may be running on a separate computer or on the host computer 10 itself. The directory-services routine maintains a data base or table of records called service records which indicate which services supply information on which subjects, where those services are located, and the service disciplines used by those services for communication. The directory-services component 192 receives the request passed from the subject mapper 180 and uses the subject parameter of that request to search through its tables for a match. That is, the directory-services component 192 searches through its service records until a service record is found indicating a particular service or services which provide information on the desired subject. This service record is then passed back to the subject mapper as symbolized by line 194. The directory-services component may find several matches if multiple services supply information regarding the desired subject.

The service record or records passed back to the subject mapper symbolized by line 194 contain many fields. Two required fields in the service records are the name of the service which provides information on the desired subject and the name of the service discipline used by that service. Other optional fields which may be provided are the name of the server upon which said service is running and a location on the network of that server.

Generally, the directory-services component will deliver all the service records for which there is a subject map, because there may not be a complete overlap in the information provided on the subject by all services. Further, each service will run on a separate server which may or may not be coupled to the client application by the same network. If such multiplicity of network paths and services exists, passing all the service records with subject matter matches back to the subject mapper provides the ability for the communications interface to switch networks or switch servers or services in the case of failure of one or more of these items.

As noted above, the subject mapper 180 functions to set up communications with all of the services providing information on the desired subject. If multiple service records are passed back from the directory-services module 192, then the subject mapper 180 will set up communications with all of these services.

Upon receipt of the service records, the subject mapper will call each identified service discipline and pass to it the subject and the service record applicable to that service discipline. Although only three service disciplines 182, 184 and 186 are shown in FIG. 14, there may be many more than three in an actual system.

In the event that the directory-services component 192 does not exist or does not find a match, no service records will be returned to the subject mapper 180. In such a case, the subject mapper will call a default service discipline and pass it and the subject and a null record.

Each service discipline is a software module which contains customized code optimized for communication with the particular service associated with that service discipline.

Each service discipline called by the subject mapper 180 examines the service records passed to it and determines the location of the service with which communications are to be established. In the particular hypothetical example being considered, assume that only one service record is returned by the directory-services module 192 and that that service record identifies the Dow Jones news service running on server 196 and further identifies service discipline A at 182 as the appropriate service discipline for communications with the Dow Jones news service on server 196. Service discipline A will then pass a request message to server 196 as symbolized by line 198. This request message passes the subject to the service and may pass all or part of the service record.

The server 196 processes the request message and determines if it can, in fact, supply information regarding the desired subject. It then sends back a reply message symbolized by line 200.

Once communications are so established, the service sends all items of information pertaining to the requested subject on a continual basis to the appropriate service discipline as symbolized by path 202. In the example chosen here, the service running on server 196 filters out only those news items which pertain to IBM for sending to service discipline at 182. In other embodiments, the server may pass along all information it has without filtering this information by subject. The communications component 30 then filters out only the requested information and passes it along to the requesting application 16. In some embodiments this is done by the daemon to be described below, and in other embodiments, it is done elsewhere such as in the information or service layers to be described below.

Each service discipline can have a different behavior. For example, service discipline B at 184 may have the following

behavior. The service running on server 196 may broadcast all news items of the Dow Jones news service on the network 14. All instances of service discipline B may monitor the network and filter out only those messages which pertain to the desired subject. Many different communication protocols are possible.

The service discipline A at 182 receives the data transmitted by the service and passes it to the named callback routine 204 in the client application 16. (The service discipline 182 was passed the name of the callback routine in the initial message from the mapper 180 symbolized by line 181.) The named callback routine then does whatever it is programmed to do with the information regarding the desired subject.

Data will continue to flow to the named callback routine 204 in this manner until the client application 16 expressly issues a cancel command to the subject mapper 180. The subject mapper 180 keeps a record of all subscriptions in existence and compares the cancel command to the various subscriptions which are active. If a match is found, the appropriate service discipline is notified of the cancel request, and this service discipline then sends a cancel message to the appropriate server. The service then cancels transmission of further data regarding that subject to the service discipline which sent the cancel request.

It is also possible for a service discipline to stand alone and not be coupled to a subject mapper. In this case the service discipline or service disciplines are linked directly to the application, and subscribe calls are made directly to the service discipline. The difference is that the application must know the name of the service supplying the desired data and the service discipline used to access the service. A database or directory-services table is then accessed to find the network address of the identified service, and communications are established as defined above. Although this software architecture does not provide data distribution decoupling, it does provide service protocol decoupling, thereby freeing the application from the necessity to know the details of the communications interface with the service with which data is to be exchanged.

More details on subject-based addressing subscription services provided by the communications interface according to the teachings of the invention are given in Section 4 of the communications interface specification given below. The preferred embodiment of the communications interface of the invention is constructed in accordance with that specification.

An actual subscribe function in the preferred embodiment is done by performing the TIB_Consume_Create library routine described in Section 4 of the specification. The call to TIB_Consume_Create includes a property list of parameters which are passed to it, one of which is the identity of the callback routine specified as My_Message_Handler in Section 4 of the specification.

In the specification, the subject-based addressing subscription service function is identified as TIBINFO. The TIBINFO interface consists of two libraries. The first library is called TIBINFO_CONSUME for data consumers. The second library is called TIBINFO_PUBLISH for data providers. An application includes one library or the other or both depending on whether it is a consumer or a provider or both. An application can simultaneously be a consumer and a provider.

Referring to FIG. 15, there is shown a block diagram of the relationship of the communications interface according to the teachings of the invention to the applications and the

network that couples these applications. Blocks having identical reference numerals to blocks in FIG. 1 provide similar functional capabilities as those blocks in FIG. 1. The block diagram in FIG. 15 shows the process architecture of the preferred embodiment. The software architecture corresponding to the process architecture given in FIG. 15 is shown in block form in FIG. 16.

The software architecture and process architecture detailed in FIGS. 15 and 16, respectively, represents an alternative embodiment to the embodiment described above with reference to FIGS. 1-14.

Referring to FIG. 15, the communications component 30 of FIG. 1 is shown as two separate functional blocks 30A and 30B in FIG. 15. That is, the functions of the communications component 30 in FIG. 1 are split in the process architecture of FIG. 15 between two functional blocks. A communications library 30A is linked with each client application 16, and a "backend communications" daemon process 30B is linked to the network 14 and to the communication library 30A. There is typically one communication daemon per host processor. This host processor is shown at 230 in FIG. 15 but is not shown at all in FIG. 16. Note that in FIG. 15, unlike the situation in FIG. 1, the client applications 16 and 18 are both running on the same host processor 230. Each client application is linked to its own copies of the various library programs in the communication libraries 30A and 96 and the form library of the data-exchange components 32 and 88. These linked libraries of programs share a common communication daemon 30B.

The communication daemons on the various host processors cooperate among themselves to insure reliable, efficient communication between machines. For subject addressed data, the daemons assist in its efficient transmission by providing low-level system support for filtering messages by subject. The communication daemons implement various communication protocols described below to implement fault tolerance, load balancing and network efficiency.

The communication library 30A performs numerous functions associated with each of the application-oriented communication suites. For example, the communication library translates subjects into efficient message headers that are more compact and easier to check than ASCII subject values. The communications library also maps service requests into requests targeted for particular service instances, and monitors the status of those instances.

The data-exchange component 32 of the communications interface according to the teachings of the invention is implemented as a library called the "form library." This library is linked with the client application and provides all the core functions of the data-exchange component. The form library can be linked independently of the communication library and does not require the communication daemon 30B for its operation.

The communication daemon serves in two roles. In the subject-based addressing mode described above where the service instance has been notified of the subject and the network address to which data is to be sent pertaining to this subject, the communication daemon 30B owns the network address to which the data is sent. This data is then passed by the daemon to the communication library bound to the client application, which in turn passes the data to the appropriate callback routine in the client application. In another mode, the communication daemon filters data coming in from the network 14 by subject when the service instances providing data are in a broadcast mode and are sending out data regarding many different subjects to all daemons on the network.

The blocks 231, 233 and 235 in FIG. 15 represent the interface functions which are implemented by the programs in the communication library 30A and the form library 32. The TIBINFO interface 233 provides subject-based addressing services by the communication paradigm known as the subscription call. In this paradigm, a data consumer subscribes to a service or subject and in return receives a continuous stream of data about the service or subject until the consumer explicitly terminates the subscription (or a failure occurs). A subscription paradigm is well suited to real-time applications that monitor dynamically changing values, such as a stock price. In contrast, the more traditional request/reply communication is ill suited to such real-time applications, since it requires data consumers to "poll" data providers to learn of changes.

The interface 235 defines a programmatic interface to the protocol suite and service comprising the Market Data Subscription Service (MDSS) sub-component 234 in FIG. 16. This service discipline will be described more fully later. The RMDP interface 235 is a service address protocol in that it requires the client application to know the name of the service with which data is to be exchanged.

In FIG. 16 there is shown the software architecture of the system. A distributed communications component 232 includes various protocol engines 237, 239 and 241. A protocol engine encapsulates a communication protocol which interfaces service discipline protocols to the particular network protocols. Each protocol engine encapsulates all the logic necessary to establish a highly reliable, highly efficient communication connection. Each protocol engine is tuned to specific network properties and specific applications properties. The protocol engines 237, 239 and 241 provide a generic communication interface to the client applications such as applications 16 and 18. This frees these applications (and the programmers that write them) from the need to know the specific network or transport layer protocols needed to communicate over a particular network configuration. Further, if the network configuration or any of the network protocols are changed such as by addition of a new local area network, gateway etc. or switching of transport layer protocols say from DECNET™ to TCP/IP™, the application programs need not be changed. Such changes can be accommodated by the addition, substitution or alteration of the protocol engines so as to accommodate the change. Since these protocol engines are shared, there is less effort needed to change the protocol engines than to change all the applications.

The protocol engines provide protocol transparency and communication path transparency to the applications thereby freeing these applications from the need to have code which deals with all these details. Further, these protocol engines provide network interface transparency.

The protocol engines can also provide value added services in some embodiments by implementing reliable communication protocols. Such value added services include reliable broadcast and reliable point to point communications as well as Reliable Multicast™ communications where communications are switched from reliable broadcast to reliable point to point when the situation requires this change for efficiency. Further, the protocol engines enhance broadcast operations where two or more applications are requesting data on a subject by receiving data directed to the first requesting application and passing it along to the other requesting applications. Prior art broadcast software does not have this capability.

The protocol engines also support efficient subject based addressing by filtering messages received on the network by

subject. In this way, only data on the requested subject gets passed to the callback routine in the requesting application. In the preferred embodiment, the protocol engines coupled to the producer applications or service instances filter the data by subject before it is placed in the network thereby conserving network bandwidth, input/output processor bandwidth and overhead processing at the receiving ends of communication links.

The distributed communication component 232 (hereafter DCC) in FIG. 16 is structured to meet several important objectives. First, the DCC provides a simple, stable and uniform communication model. This provides several benefits. It shields programmers from the complexities of: the distributed environment; locating a target process; establishing communications with this target process; and determining when something has gone awry. All these tasks are best done by capable communications infrastructure and not by the programmer. Second, the DCC reduces development time not only by increasing programmer productivity but also by simplifying the integration of new features. Finally, it enhances configurability by eliminating the burden on applications to know the physical distribution on the network of other components. This prevents programmers from building dependencies in their code on particular physical configurations which would complicate later reconfigurations.

Another important objective is the achievement of portability through encapsulation of important system structures. This is important when migrating to a new hardware or software environment because the client applications are insulated from transport and access protocols that may be changing. By isolating the required changes in a small portion of the system (the DCC), the applications can be ported virtually unchanged and the investment in the application software is protected.

Efficiency is achieved by the DCC because it is coded on top of less costly "connectionless" transport protocol in standard protocol suites such as TCP/IP and OSI. The DCC is designed to avoid the most costly problem in protocols, i.e., the proliferation of data "copy" operations.

The DCC achieves these objectives by implementing a layer of services on top of the basic services provided by vendor supplied software. Rather than re-inventing basic functions like reliable data transfer or flow-control mechanisms, the DCC shields applications from the idiosyncracies of any particular operating system. Examples include the hardware oriented interfaces of the MS-DOS environment, or the per-process file descriptor limit of UNIX. By providing a single unified communication toll that can be easily replicated in many hardware and software environments, the DCC fulfills the above objectives.

The DCC implements several different transmission protocols to support the various interaction paradigms, fault-tolerance requirements and performance requirements imposed by the service discipline protocols. Two of the more interesting protocols are the reliable broadcast and intelligent multicast protocols.

Standard broadcast protocols are not reliable and are unable to detect lost messages. The DCC reliable broadcast protocols ensure that all operational hosts either receive each broadcast message or detects the loss of the message. Unlike many so-called reliable broadcast protocols, lost messages are retransmitted on a limited, periodic basis.

The Intelligent Multicast™ protocol provides a reliable datastream to multiple destinations. The novel aspect of this protocol is that it can switch dynamically from point-to-

point transmission to broadcast transmission in order to optimize the network and processor load. The switch from point-to-point to broadcast (and vice-versa) is transparent to higher-level protocols. This transport protocol allows the support of a much larger number of consumers than would be possible using either point-to-point or broadcast alone. The protocol is built on top of other protocols with the DCC.

Currently, all DCC protocols exchange data only in discrete units, i.e., messages (in contrast to many transport protocols). The DCC guarantees that the messages originating from a single process are received in the order sent.

The DCC contains fault tolerant message transmission protocols that support retransmission in the event of a lost message. The DCC software guarantees "at-most-once" semantics with regard to message delivery and makes a best attempt to ensure "exactly-once" semantics.

The DCC has no exposed interface for use by application programmers.

The distributed component 232 is coupled to a variety of service disciplines 234, 236 and 238. The service discipline 234 has the behavior which will herein be called Market Data Subscription Service. This protocol allows data consumers to receive a continuous stream of data, fault tolerant of failures of individual data sources. This protocol suite provides mechanisms for administering load-balancing and entitlement policies.

The MDSS service discipline is service oriented in that applications calling this service discipline through the RMDP interface must know the service that supplies requested data. The MDSS service discipline does however support the subscription communication paradigm which is implemented by the Subject Addressed Subscription Service (SASS) service discipline 238 in the sense that streams of data on a subject will be passed by the MDSS service discipline to the linked application.

The MDSS service discipline allows data consumer applications to receive a continuous stream of data, tolerant of failures of individual data sources. This protocol suite 234 also provides mechanisms for load balancing and entitlement policy administration where the access privileges of a user or application are checked to insure a data consumer has a right to obtain data from a particular service.

Two properties distinguish the MDSS service discipline from typical client server protocols. First, subscriptions are explicitly supported whereby changes to requested values are automatically propagated to requesting applications. Second client applications request or subscribe to a specific service (as opposed to a particular server and as opposed to a particular subject). The MDSS service discipline then forwards the client application request to an available server. The MDSS service discipline also monitors the server connection and reestablishes it if the connection fails using a different server if necessary.

The MDSS service discipline implements the following important objectives.

Fault tolerance is implemented by program code which performs automatic switchover between redundant services by supporting dual or triple networks and by utilizing the fault tolerant transmission protocols such as reliable broadcast implemented in the protocol engines. Recovery is automatic after a server failure.

Load balancing is performed by balancing the data request load across all operating servers for a particular service. The load is automatically rebalanced when a server fails or recovers. In addition, the MDSS supports server

assignment policies that attempts to optimize the utilization of scarce resources such as "slots" in a page cache or bandwidth across an external communication line.

Network efficiency is implemented by an intelligent multicast protocol implemented by the distributed communication daemon 30B in FIG. 15. The intelligent multicast protocol optimizes limited resources of network and I/O processor bandwidth by performing automatic, dynamic switchover from point to point communication protocols to broadcast protocols when necessary. For example, Telerate page 8 data may be provided by point to point distribution to the first five subscribers and then switch all subscribers to broadcast distribution when the sixth subscriber appears.

The MDSS service discipline provides a simple, easy-to-use application development interface that masks most of the complexity of programming a distributed system, including locating servers, establishing communication connections, reacting to failures and recoveries and load balancing.

The core functions of the MDSS service discipline are: get, halt and derive. The "get" call from a client application establishes a fault-tolerant connection to a server for the specified service and gets the current value of the specified page or data element. The connection is subscription based so that updates to the specified page are automatically forwarded to the client application. "Halt" stops the subscription. "Drive" sends a modifier to the service that can potentially change the subscription.

The MDSS service discipline is optimized to support page oriented services but it can support distribution of any type data.

The service discipline labeled MSA, 236, has yet a different behavior. The service discipline labeled SASS, 238, supports subject-based address subscription services.

The basic idea behind subject based addressing and the SASS service discipline's (hereafter SASS) implementation of it is straightforward. Whenever an application requires data, especially data on a dynamically changing value, the application simply subscribes to it by specifying the appropriate subject. The SASS then maps this subject request to one or more service instances providing information on this subject. The SASS then makes the appropriate communication connections to all the selected services through the appropriate one or more protocol engines necessary to communication with the servicer or servers providing the selected service or services.

Through the use of subject based addressing, information consumers can request information in a way that is independent of the application producing the information. Hence, the producing application can be modified or supplanted by a new application providing the same information without affecting the consumers of the information.

Subject based addressing greatly reduces the complexities of programming a distributed application in three ways. First, the application requests information by subject, as opposed to by server or service. Specifying information at this high level removes the burden on applications of needing to know the current network address of the service instances providing the desired information. It further relieves the application of the burden of knowing all the details of the communication protocols to extract data from the appropriate service or services and the need to know the details of the transport protocols needed to traverse the network. Further, it insulates the client applications from the need for programming changes when something else changes like changes in the service providers, e.g., a change from IDN to Ticker 3 for equity prices. All data is provided

through a single, uniform interface to client applications. A programmer writing a client application needing information from three different services need not learn three different service specific communication protocols as he or she would in traditional communication models. Finally, the SASS automates many of the difficult and error prone tasks such as searching for an appropriate service instance and establishing a correct communication connection.

The SASS service discipline provides three basic functions which may be invoked through the user interface.

"Subscribe" is the function invoked when the consumer requests information on a real-time basis on one or more subjects. The SASS service discipline sets up any necessary communication connections to ensure that all data matching the given subject(s) will be delivered to the consumer application. The consumer can specify that data be delivered either asynchronously (interrupt-driven) or synchronously.

The producer service will be notified of the subscription if a registration procedure for its service has been set up. This registration process will be done by the SASS and is invisible to the user.

The "cancel" function is the opposite of "subscribe". When this function is invoked, the SASS closes down any dedicated communication channel and notifies the producer service of the cancellation if a registration procedure exists.

The "Receive" function and "callback" function are related functions by which applications receive messages matching their subscriptions. Callbacks are asynchronous and support the event driven programming style. This style is well suited for applications requiring real time data exchange. The receive function supports a traditional synchronous interface for message receipt.

A complementary set of functions exists for a data producer. Also, applications can be both data producers and data consumers.

Referring to FIG. 17 there is shown a typical computer network situation in which the teachings of the invention may be profitably employed. The computer network shown is comprised of a first host CPU 300 in Houston coupled by a local area network (hereafter LAN) 302 to a file server 304 and a gateway network interconnect circuit 306. The gateway circuit 306 connects the LAN 302 to a wide area network (hereafter WAN) 308. The WAN 308 couples the host 300 to two servers 310 and 312 providing the Quotron and Marketfeed 2000 services, respectively, from London and Paris, respectively. The WAN 308 also couples the host 300 to a second host CPU 314 in Geneva and a server 316 in Geneva providing the Telerate service via a second LAN 318. Dumb terminal 320 is also coupled to LAN 318.

Typically the hosts 300 and 314 will be multitasking machines, but they may also be single process CPU's such as computers running the DOS or PC-DOS operating systems. The TIB communication interface software supplied herewith as Appendix A embodies the best mode of practicing the invention and is ported for a Unix based multitasking machine. To adapt the teachings of the invention to the DOS or other single task environments requires that the TIB communication daemon 30B in the process architecture be structured as an interrupt driven process which is invoked, i.e., started upon receipt of a notification from the operating system that a message has been received on the network which is on a subject to which one of the applications has subscribed.

The LAN's 302 and 318, WAN 308 and gateway 306 may each be of any conventional structure and protocol or any new structure and protocol developed in the future so long

as they are sufficiently compatible to allow data exchange among the remaining elements of the system. Typically, the structures and protocols used on the networks will be TCP/IP, DECNET™, ETHERNET™, token ring, ARPANET and/or other digital pack or high speed private line digital or analog systems using hardware, microwave or satellite transmission media. Various CCITT recommendations such as X.1, X.2, X.3, X.20, X.21, X.24, X.28, X.29, X.25 and X.75 suggest speeds, user options, various interface standards, start-stop mode terminal handling, multiplex interface for synchronous terminals, definitions of interface circuits and packet-network interconnection, all of which are hereby incorporated by reference. A thorough discussion of computer network architecture and protocols is included in a special issue of IEEE Transactions on Communications, April 1980, Vol. COM-28, which also is incorporated herein by reference. Most digital data communication is done by characters represented as sequences of bits with the number of bits per character and the sequence of 0's and 1's that correspond to each character defining a code. The most common code is International Alphabet No. 5 which is known in the U.S. as ASCII. Other codes may also be used as the type of code used is not critical to the invention.

In coded transmission, two methods of maintaining synchronism between the transmitting and receiving points are commonly used. In "start-stop" transmission, the interval between characters is represented by a steady 1 signal, and the transmission of a single 0 bit signals the receiving terminal that a character is starting. The data bits follow the start bit and are followed by a stop pulse. The stop pulse is the same as the signal between characters and has a minimum length that is part of the terminal specification. In the synchronous method, bits are sent at a uniform rate with a synchronous idle pattern during intervals when no characters are being sent to maintain timing. The synchronous method is used for higher speed transmission.

Protocols as that term is used in digital computer network communication are standard procedures for the operation of communication. Their purpose is to coordinate the equipment and processes at interfaces at the ends of the communication channel. Protocols are considered to apply to several levels. The International Organization for Standardization (ISO) has developed a seven level Reference Model of Open System Interconnection to guide the development of standard protocols. The seven levels of this standard hereafter referred to as the ISO Model and their functions are:

- (1) Application: Permits communication between applications. Protocols here serve the needs of the end user.
- (2) Presentation: Presents structured data in proper form for use by application programs. Provides a set of services which may be selected by the application layer to enable it to interpret the meaning of data exchanged.
- (3) Session: Sets up and takes down relationships between presentation entities and controls data exchange, i.e., dialog control.
- (4) Transport: Furnishes network-independent transparent transfer of data. Relieves the session layer from any concern with the detailed way in which reliable and cost-effective transfer of data is achieved.
- (5) Network: Provides network independent routing, switching services.
- (6) Data Link: Gives error-free transfer of data over a link by providing functional and procedural means to establish, maintain and release data links between network entities.

- (7) Physical: Provides mechanical, electrical, functional and procedural characteristics to establish, maintain, and release physical connections, e.g., data circuits between data link entities.

Some data link protocols, historically the most common, use characters or combinations of characters to control the interchange of data. Others, including the ANSI Advanced Data Communication Control Procedure and its subsets use sequences of bits in predetermined locations in the message to provide the link control.

Packet networks were developed to make more efficient use of network facilities than was common in the circuit-switched and message-switched data networks of the mid-60's. In circuit-switched networks, a channel was assigned full time for the duration of a call. In message-switched networks, a message or section of a serial message was transmitted to the next switch if a path (loop or trunk) was available. If not, the message was stored until a path was available. The use of trunks between message switches was often very efficient. In many circuit-switched applications though, data was transmitted only a fraction of the time the circuit was in use. In order to make more efficient use of facilities and for other reasons, packet networks came into existence.

In a packet network, a message from one host or terminal to another is divided into packets of some definite length, usually 128 bytes. These packets are then sent from the origination point to the destination point individually. Each packet contains a header which provides the network with the necessary information to handle the packet. Typically, the packet includes at least the network addresses of the source and destination and may include other fields of data such as the packet length, etc. The packets transmitted by one terminal to another are interleaved on the facilities between the packets transmitted by other users to their destinations so that the idle time of one source can be used by another source. Various network contention resolution protocols exist to arbitrate for control of the network by two or more destinations wishing to send packets on the same channel at the same time. Some protocols utilize multiple physical channels by time division or frequency multiplexing.

The same physical interface circuit can be used simultaneously with more than one other terminal or computer by the use of logical channels. At any given time, each logical channel is used for communication with some particular addressee; each packet includes in its header the identification of its logical channel, and the packets of the various logical channels are interleaved on the physical-interface circuit.

At the destination, the message is reassembled and formatted before delivery to the addressee process. In general, a network has an internal protocol to control the movement of data within the network.

The internal speed of the network is generally higher than the speed of any terminal or node connected to the network.

Three methods of handling messages are in common use. "Datagrams" are one-way messages sent from an originator to a destination. Datagram packets are delivered independently and not necessarily in the order sent. Delivery and nondelivery notifications may be provided. In "virtual calls", packets are exchanged between two users of the network; at the destination, the packets are delivered to the addressee process in the same order in which they were originated. "Permanent virtual circuits" also provide for exchange of packets between two users on a network. Each assigns a logical channel, by arrangement with the provider of the

network, for exchange of packet with the other. No setup or clearing of the channel is then necessary.

Some packet networks support terminals that do not have the internal capability to format messages in packets by means of a packet assembler and disassembler included in the network.

The earliest major packet network in the U.S. was ARP-NET, set up to connect terminals and host computers at a number of universities and government research establishments. The objective was to permit computer users at one location to use data or programs located elsewhere, perhaps in a computer of a different manufacturer. Access to the network is through an interface message processor (IMP) at each location, connected to the host computer(s) there and the IMP at other locations. IMP's are not directly connected to each other IMP. Packets routed to destination IMP's not connected directly to the source IMP are routed through intervening IMP's until they arrive at the destination process. At locations where there is no host, terminal interface processors are used to provide access for dumb terminals.

Other packet networks have subsequently been set up worldwide, generally operating in the virtual call mode.

In early packet networks, routing of each packet in a message is independent. Each packet carries in its header the network address of the destination as well as a sequence number to permit arranging of the packets in the proper order at the destination. Networks designed more recently use a "virtual circuit" structure and protocol. The virtual circuit is set up at the beginning of a data transmission and contains the routing information for all the packets of that data transmission. The packets after the first contain the designation of the virtual circuit in their headers. In some networks, the choice of route is based on measurements received from all other nodes, of the delay to every other node on the network. In still other network structures, nodes on the network are connected to some or all the other nodes by doubly redundant or triply redundant pathways.

Some networks such as Dialog, Tymshare and Telenet use the public phone system for interconnection and make use of analog transmission channels and modems to modulate digital data onto the analog signal lines.

Other network structures, generally WAN's, use microwave and/or satellites coupled with earth stations for long distance transmissions and local area networks or the public phone system for local distribution.

There is a wide variety of network structures and protocols in use. Further, new designs for network and transport protocols, network interface cards, network structures, host computers and terminals, server protocols and transport and network layer software are constantly appearing. This means that the one thing that is constant in network design and operation is that it is constantly changing. Further, the network addresses where specific types of data may be obtained and the access protocols for obtaining this data are constantly changing. It is an object of the communication interface software of the invention to insulate the programmer of application programs from the need to know all the networks and transport protocols, network addresses, access protocols and services through which data on a particular subject may be obtained. By encapsulating and modularizing all this changing complexity in the interface software of the invention, the investment in application programs may be protected by preventing network topology or protocol dependencies from being programmed into the applications. Thus, when something changes on the network, it is not necessary to reprogram or scrap all the application programs.

The objectives are achieved according to the teachings of the invention by network communications software having a three-layer architecture, hereafter sometimes called the TIB™ software. In FIG. 17, these three layers are identified as the information layer, the service layer and the distributed communication layer. Each application program is linked during the compiling and linking process to its own copy of the information layer and the service layer. The compiling and linking process is what converts the source code of the application program to the machine readable object code. Thus, for example, application program 1, shown at 340, is directly linked to its own copy of layers of the software of the invention, i.e., the information layer 342 and the service layer 344. Likewise application 2, shown at 346 is linked to its own copies of the information layer 348 and the service layer 350. These two applications share the third layer of the software of the invention called the distributed communication layer 352. Typically there is only one distributed communication layer per node (where a node is any computer, terminal or server coupled to the network) which runs concurrently with the applications in multitasking machines but which could be interrupt drivers in nonmultitasking environments.

The second host 314 in Geneva in the hypothetical network of FIG. 17 is running application program 3, 354. This application is linked to its copies of the information layer 356 and the service layer 358. A concurrently running distributed communication layer 360 in host 2 is used by application 354.

Each of the servers 310, 312 and 316 have a data producer versions of the 3 layer TIB™ software. There is a data consumer version of the TIB™ software which implements the "subscribe" function and a data producer version which implements the "publish" function. Where a process (a program in execution under the UNIX™ definition) is both a data consumer and a data publisher, it will have libraries of programs and interface specifications for its TIB™ software which implement both the subscribe and publish functions.

Each of the hosts 300 and 314 is under the control of an operating system, 370 and 372, respectively, which may be different. Host 1 and host 2 may also be computers of different manufacturers as may servers 310, 312 and 316. Host 1 has on-board shared memory 374 by which applications 340 and 346 may communicate such as by use of a UNIX™ pipe or other interprocess communication mechanism. Host 2 utilizes memory 378.

In a broad statement of the teachings of the invention, the information layer, such as 342, encapsulates the TIBINFO™ interface functionality, and the subject-based addressing functionality of the TIB™ software communication library 30A of FIGS. 15 and 16. The TIBINFO interface is defined in Section 4 of the software specification below. TIBINFO defines a programmatic interface by which applications linked to this information layer may invoke the protocols and services of the Subject-Addressed Subscription Service (SASS) component.

FIG. 18 clarifies the relationships between the process architecture of FIG. 15, the software architecture of FIG. 16 and the 3 layers for the TIB™ software shown in FIG. 17. In FIG. 15, the communications library 30A is a library of programs which are linked to the application 16 which provide multiple functions which may be called upon by the RMDP and TIBINFO interfaces. Subject-Addressed Subscription Services are provided by subject mapper programs and service discipline programs in the component labeled 30A/30B. This component 30A/30B also includes library

programs that provide the common infrastructure program code which supports, i.e., communicates with and provides data to, the protocol engine programs of the TIB communication daemon 30B.

The TIBINFO interface is devoted to providing a programmatic interface by which linked client applications may start and use subject-addressed subscriptions for data provided by data producers on the network wherever they may be.

The RMDP interface provide the programmatic interface by which subscriptions may be entered and data received from services on the network by linked client applications which already know the names of the services which supply this data. The communication library 30A in FIG. 15 supplies library programs which may be called by linked client applications to implement the Market-Data-Subscription Service (MDSS). This function creates subscription data requested by service names by setting up, with the aid of the daemon's appropriate protocol engine, a reliable communication channel to one or more servers which supply the requested data. Failures of individual servers are transparent to the client since the MDSS automatically switches to a new server which supplies the same services using the appropriate protocol engine. The MDSS also automatically balances the load on servers and implements entitlement functions to control who gets access to a service. The reliable communication protocols of the DCC library 30A/30B such as intelligent multicast and reliable broadcast and the protocol engines of the daemon 30B are invoked by the MDSS library programs. More details are given in Section 3 of the TIB™ Specification given below.

Referring to FIG. 19, which is comprised of FIGS. 19A and 19B, there is shown a flow chart for the process carried out, inter alia, at each of the 3 layers on the subscriber process side for entering a subscription on a particular subject and receiving data on the subject. Step 400 represents the process of receiving a request from a user for data on a particular subject. This request could come from another process, another machine or from the operating system in some embodiments. For purposes of this example assume that the request comes to application 1 in FIG. 17 from a user.

Application 1 (on the application layer or layer 1 of the ISO Model) then sends a "subscribe request" to information layer 342 in FIG. 17. This process is represented by step 402 in FIG. 19A. This subscribe request is entered by calling the appropriate library program in the linked library of programs which includes the TIB-INFO interface. This subroutine call passes the subject on which data is requested and a pointer to the callback routine in the requesting process that the TIB-INFO library program on the information layer is to call when messages are received on this subject.

The information layer 342 encapsulates a subject-to-service discipline mapping function which provides architectural decoupling of the requesting process as that term is defined in the glossary herein. Referring to steps 404 and 406 in FIG. 19A and to FIG. 17, the input to the information layer is the subject and the output is a call to a service discipline on the service layer 344 in FIG. 17. The information layer includes the TIB-INFO interface and all library programs of the linked communications library 30A in FIG. 15 involved with subject-to-service mapping. The information layer maps the subject to the service or services which provide data on this subject as symbolized by step 404 and then maps this service or services to one or more service disciplines that encapsulate communication protocols to communicate with these services. This information layer

then coordinates with the service discipline to assign a "TIB channel" as symbolized by step 410.

A "TIB channel" is like an "attention: Frank Jones" line on the address of a letter. This TIB channel data is used to route the message to the process which requested data on whatever subject is assigned to that TIB channel. Each subject is assigned a TIB channel when a subscription is entered. There is a subscription list that correlates the subscribing processes, their network addresses, the subjects subscribed to and the TIB channel numbers assigned to these subjects. Data on this list is used by the daemon to route messages received at its port address to the proper requesting process. This list is also used on the data publisher side to cause messages on particular subjects to be routed to the port address of the machine on which the requesting process is running. The communication layer of the TIB software associated with the service writes the channel number data in the headers of packets from messages on particular subjects before these packets are transmitted on the network. At the receiver side, the TIB channel data in the header causes proper routing of the packet to the requesting process. The TIB channel abstraction and the routing function it implies is performed by the DCC library portion 30A/30B in FIG. 18 which is linked to each requesting process.

Assuming there are two such services, these services are then mapped by the service disciplines on the service layer to the servers that provide these services as symbolized by step 412.

In one embodiment, the information layer selects and calls only one of the service discipline subroutines in the service layer as symbolized by step 406. The service discipline then runs and assigns a TIB channel to the subscription subject as symbolized by step 410. The call from the information layer also passes the pointer to a callback routine in the information layer to be called when messages on the subject arrive.

In alternative embodiments, the information layer may call all the service disciplines identified in the subject-to-service discipline mapping process so as to set up communication links with all the services.

In some embodiments, the names of alternative services and alternative servers are passed to the selected service discipline or directly to the distributed communication layer by the information layer for use in setting up alternate communication links. This allows the distributed communication layer to set up an alternate communication link to another server in case of failure of the selected server or for simultaneous communication link setup to increase the throughput of the network. In still other embodiments, the requesting process can call the service layer directly and invoke the appropriate service discipline by doing the subject-to-service discipline mapping in the application itself. The data regarding alternate services and servers can be passed by calling a library subroutine in the DCC library of block 30A/30B in FIG. 18 which runs and stores the data regarding the alternates.

In alternative embodiments, the information layer may assign the TIB channel to each subject or the service layer may assign the TIB channel acting alone without coordinating with the information layer. Step 410 represents the embodiments where the service discipline assigns the TIB channel number by coordinating with the information layer. Messages sent by data provider processes will have the assigned subject channel data included as part of their header information.

TIB channels are used by the communication layer (DCC) for filtering and routing purposes. That is, the daemon 30B

in FIG. 15 and protocol engines 30B in FIG. 18 know when a message arrives at the daemon's port address having particular TIB channel data in the header that there are outstanding subscriptions for data on this subject. The daemon process knows the channels for which there are outstanding subscriptions because this information is sent to the communication layer by the service layer. The daemon 30B stores data received from the service discipline regarding all TIB channels having open subscriptions. The daemon then sends any message on a subject having an open subscription to processes at that port address which have subscribed to messages on that subject. The daemon does not know what the subject is but it does know there is a match between TIB channels having open subscriptions and subjects of some of the incoming messages.

Each node coupled to the computer network of FIG. 17 such as host 300 has one network interface card and one port address. This port address may be assigned a "logical channel" in some networks for multiplexing of the network card by multiple processes running simultaneously on the same host. These port addresses may sometimes hereafter also be referred to as network addresses. How data gets back and forth between network addresses is the responsibility of the communication layers such as layer 352 in FIG. 17 which invokes the transport layer, network layer, data link layer and physical layer functionalities of the operating systems 370 and 372, network interface cards (not shown) and other circuits on the network itself such as might be found in gateway 306 and WAN 308.

The service layer, information layer and communication layer are layers which are "on top" of the ISO model layers, i.e., they perform services not performed on any layer of the ISO model or they perform "value added" services as an adjunct to services performed on an ISO model layer.

The purpose of the service layer is, among other things, to provide service decoupling as that term is defined in the glossary herein. Service decoupling frees the application of the need to know the details of how to communicate which services. To perform this function, the service layer includes a program or function to map the selected service to all servers that provide this service and pick one (step 412 in FIG. 19). The service layer then maps the selected server to a protocol engine that encapsulates the communication procedures or protocols necessary to traverse the network, i.e., set up a data link regardless of the path that needs to be followed through the network to get to this server, and communicate with the selected server regardless of what type of machine it is, what type of network and network card it is coupled to and what operating system this server runs. This process is symbolized by step 414.

In alternative embodiments, the application or subscribing process may call the protocol engine directly by having done its own subject based addressing and encapsulating its own communication protocol. In other alternative embodiments, the service layer will select all the servers supplying a service and request the communication layer to set up data links with all of them simultaneously to increase the throughput of the network or to use one server and switch to another upon failure of the selected server.

Normally all services that provide a selected service are assumed to use the same communication protocol so a single service discipline can communicate with them all. However, if different instances of the same services or different services providing data on the same subjects use different communication protocols, the teachings of the invention contemplate subclasses of service disciplines. This means that the information layer will call a generic service disci-

pline which contains code which can be shared by all subclasses of this service discipline to do common functions such as subscribe or cancel which are done the same way on all servers that provide this service. The generic service discipline will then map the subscription request to one or more of the different servers that provide the service. The service discipline(s) code which encapsulates the communication procedure peculiar to the selected server(s) is then called and runs to finish the process of setting up the subscription data stream with the selected server(s).

The output of the service layer is a request to the communication layer to the effect, "set up a communication link by whatever means you deem most appropriate with the following server and services on the following subject channel." The service layer also sends a pointer to the service layer callback routine which will handle messages or packets on the requested subject. This process is symbolized by step 416. In some embodiments the network addresses of all servers that run service processes supplying data on the requested subject are passed to a DCC library program which stores them for use in providing a reliable communication link by providing failure recovery in case the selected server crashes.

Step 418 represents the process where the selected protocol engine sets up the requested data link by invoking selected functions of the transport layer protocols encapsulated in the operating system. These protocols invoke other communication protocols on the network, data link and physical layers so as to set up the requested data link and log onto the service as symbolized by step 420. The service layer service discipline usually then sends a message to the service notifying it of the subscription and the TIB channel assigned to this TIB as symbolized by step 422. The subject channel is noted by the information service and/or communication layer of the TIB interface software linked to the service. This allows the TIB channel data to be added to the packet headers of transmitted packets on the subject of interest. This subscription message starts the flow of data in some embodiments, while in other embodiments, the flow of data starts when the data link to the server is first established.

In some embodiments, a single subscription may necessitate calling multiple services, so the information layer may map the subject to multiple service disciplines. These in turn map the request to multiple protocol engines which simultaneously set up data links to the multiple services.

In some alternative embodiments, the service disciplines talk directly to the transport layer and encapsulate the protocols necessary to communicate on the current network configuration. In these embodiments, the service layer may filter incoming messages by subject before calling the callback routine in the information layer.

On small networks, an alternate embodiment is to broadcast subscription requests to particular subjects on the network. Services coupled to the network listen to these broadcasts and send messages on the subjects of interest to the port addresses identified in the broadcasts. These messages are then directed by the DCC layer at the port address to the requesting process in the manner described elsewhere herein.

In alternative embodiments, the service layer also performs other functions such as: regulating access to certain services; session management in the traditional sense of the session layer of the ISO model; replication management of replicated services and servers; failure/recovery management in case of failure of a service; distribution management; load balancing to prevent one server or service from being inequitably loaded with data requests when other

services/servers can fill the need; or, security functions such as providing secure, encoded communications with a server. Of particular importance among these alternate embodiments are the embodiments which encapsulate service recovery schemes on the service layer. In these embodiments, when a server goes down, a recovery scheme to obtain the same data elsewhere encapsulated in the appropriate service discipline is run to re-establish a new data link to an alternate server as symbolized by step 424.

In the preferred embodiment, the service discipline assigns the TIB channel to the subject and picks the protocol engine to use in terms of the characteristics of the server and the service to be accessed and the network and network protocol to be used, and in light of the degree of reliability necessary.

The daemon 30B in FIGS. 15 and 18 can include many different protocol engines, each of which has different characteristics. For example there may be a protocol engine for point-to-point communication between nodes having Novell network interface cards and using the Novell protocol and a protocol engine for point-to-point communications between nodes using the TCP and UDP protocols and associated network interface cards. There may also be a protocol engine for communication with high speed data publishers using reliable broadcast, and a protocol engine for either point-to-point or reliable broadcast communication using the Intelligent Multicast™ protocol. There can be as many protocol engines as there are options for communication protocols, types of servers and services and reliability options, and more can be added at any time.

Further, some of the service disciplines may be procedures for communicating with other processes on the same machine such as the operating system or another application or directly with a user through a terminal. More service disciplines can be added at any time to communicate with new sources of information.

A service discipline, when it receives a subscription request may open a specific TIB channel for that subject or allow any arbitrary TIB channel to be used.

The selected service discipline or disciplines pick the protocol engine that has the right characteristics to efficiently communicate with the selected service by calling a DCC library program. The DCC library program updates the subscription list with the new subscription and channel data and sends a message to the selected protocol engine via shared memory or some other interprocess transfer mechanism. If the host is not multitasking, the daemon will be caused to run by an interrupt generated by the DCC library program. The message to the selected protocol engine will be as previously described and will include the identity of the selected server. The protocol engine will map the identity of this server to the network address of the server and carry out the communication protocol encapsulated within the selected protocol engine to set up the data link. Some of these protocols are value added protocols to, for example, increase the reliability of generic transport layer broadcast protocols or to do intelligent multicasting. These value added protocols will be detailed below. This step is symbolized by step 426.

The distributed communication layers 352 and 360, function to provide configuration decoupling. This eliminates the need for the requesting process to know how to do various communication protocols such as TCP, UDP, broadcast etc and to have code therein which can implement these protocols. The protocol engines implement various communication protocols and the DCC library implements the notion of TIB channels and performs routing and filtering by subject

matter based upon the TIB channel data in the packet headers of incoming packets. The protocol engine for communicating using the UDP protocol also does message disassembly into packets on the service or transmit side and packet reassembly into complete messages on the subscribing process or receive side. This is a value added service since the UDP transport protocol does not include these disassembly and reassembly functions. The TCP transport protocol includes these message disassembly and packet reassembly functions so the protocol engine that invokes this transport layer function need not supply these type value added services.

In some embodiments of the invention, the UDP protocol engine adds sequence numbers and data regarding how many packets comprise each complete message to the packet headers. This allows the daemon or DCC library of the receiving process TIB communication layer to check the integrity of the message received to insure that all packets have been received.

As data packets come in from the network, they are passed up through the DCC library, service layer and information layer to the subscribing process. The service layer in some embodiments may filter the incoming messages by subject matter instead of having this filtering done by the daemon or the DCC library as in other embodiments. In still other embodiments, the filtering by subject matter is done by the information layer.

In some embodiments, the service layer also performs data formatting by calling programs in the TIB FORMS interface 231 in FIG. 15 or the TIB Forms Library 32 in FIG. 1.

In some embodiments, the subject based addressing is done by collecting all the information a subscribing process could ever want in a gigantic data base and organizing the data base by subject matter with updates as data changes. The service layer would then comprise routines to map the subject request to data base access protocols to extract data from the proper areas of the data base. The communication layer in such embodiments maps incoming update data to update protocols to update the appropriate data in the data base.

The preferred embodiment implements the more powerful notion of allowing the data sources to be distributed. This allows new servers and services to be coupled to the system without causing havoc or obsolescence with all the existing application software. The use of the information, service and communication layers of the TIB software according to the teachings of the invention provides a very flexible way of decoupling the application software from the ever changing network below it.

In the preferred embodiment, the filtering by subject matter for point-to-point protocols is done by the TIB software on the transmit side. Note that in FIG. 17, the servers 310, 312 and 316 are decoupled from the network by TIB interface software symbolized by the blocks marked "TIB IF". Terminal 320 is also decoupled in the same manner and can be a service for manual entry of data by the user. Specifically, this filtering is done by the information layer bound to the service which is publishing the data. For a service that is using the broadcast transport protocol, the TIB communication layer at the network addresses receiving the broadcast would filter out all messages except those having subjects matching open subscriptions by comparing the TIB channel data to the channel data for open subscriptions listed in the subscription table based upon subscription data generated by the information layer and TIB channel data generated by the service layer. Note that where a service

simply broadcasts data, the service discipline for accessing that service can be as simple as "listen for data arriving at the following network address and filter out the messages on other than the subscribed subject." The service discipline would then format the data properly by invoking the proper function in the TIB Forms Library and pass the data through the information layer to the requesting process.

The use of the communication layer allows all the network configuration parameters to be outside the applications and subject to revision by the system administrator or otherwise when the network configuration changes. This insulates the application software from the network interface and provides a functionality similar to and incorporating at least all the functionality of the ISO Model network layer.

Note also that in some embodiments, the functionality of the information, service and communication layers could also be easily implemented in hardware rather than the software of the preferred embodiment. The service and communication layers implement most of the functionality the ISO Model Network, Data Link and Physical layers plus more.

In some embodiments, the distributed communication layer only receives a general request from the service layer to set up a data link and decides on its own which is the most efficient protocol to use. For example, the DCC may receive 5 separate subscriptions for the same information. The DCC may elect on its own to set up 5 separate data links or bundle the requests, set up one data link and distribute the arriving message by interprocess transfers to each of the 5 requesting processes. In other embodiments, the DCC may act on its own to decide which protocol to use, but may accept things from the service layer such as, "I want this fast" or "I want this reliable". In the latter case, the communication layer may elect to send two subscriptions for the same information to two different services or may set up two different links to the same service by different network paths.

In the preferred embodiment, the DCC library portion of the communication library serves the sole function of determining how to best get data from one network address to another. All replication management and failure recovery protocols are encapsulated in the service disciplines.

Referring to FIG. 20, comprised of FIGS. 20A and 20B, there is shown a flow chart for the processing involved at the three layers of the TIB software on the transmit side in creating a subscription data stream at a publishing process or service and sending it down through the TIB software and across the network to the subscribing process.

Step 430 represents the process whereby the selected service receives a message from a subscribing process and initiates a data stream. Each service such as the Quotron service running on server 310 in FIG. 17 and the Marketfeed 2000 and Telerate services running on servers 312 and 316, respectively, are decoupled from the network by a version of the three-layer architecture TIB software according to the teachings of the invention. This is symbolized by the blocks marked TIB IF in these server boxes which stands for TIB interface software.

The TIB interface for each service decouples the service from any requirement to have functionality capable of supporting filtering or subject based addressing. Thus, if a service is designed to broadcast all equity prices on the American Stock Exchange and Over-the-Counter market, but the subscription is simply for IBM equity prices, the service responds as it always has and need not have a function to filter out only IBM equity prices. The service discipline for this type service will be adapted to filter out all messages except IBM equity prices in response to such a subscription request.

Another service like Telerate which publishes many different pages organized by subject matter, e.g., a page on Government T-Bill rates, a page on long term corporate bond rates etc., will be able to accept a subscription for only a specific page and may be able to accept special commands to cause the service to publish only specific columns on a particular page. In such a case, the service layer bound to such a service will include a service discipline which receives subscription requests by subject and filters messages from a broadcast that do not pertain to a subject having an open subscription.

Step 430 also represents the process of the service calling the TIB-Publish function of the information layer TIB-INFO library and starting the flow of data toward the subscribing process. The service need not have any of its own ability to filter by subject. The subscription request it receives is in the "native tongue" that this service understands because it is formatted and sequenced in that fashion by the service discipline of the subscribing process.

Most of the filtering by subject matter is done by the service disciplines, but where this filtering is done depends upon the type of service. Some services publish only one type of data so everything such a publisher puts out is of interest to the subscribing process. For example, assume that the service accessed is the real time clock 371 in FIG. 17 which puts out only the current time, and assume that the subject of the subscription is "give me the time of day". In such a case, the service discipline is very simple and no filtering need occur. Such a service discipline can be simply a protocol to determine how to communicate the data to the requesting process and what TIB channel to assign to it.

The fact that the service starts sending data in whatever manner such a service normally sends data is symbolized by step 432. Thus, if the service is Telerate, it can send the page image and updates for any one of a number of different pages and it understands a subscription for only one of its many pages whereas the Quotron service would not understand a subscription for only IBM equity prices. The various service disciplines of the service layer provide, inter alia, the necessary functionality which the service does not have.

Step 432 assumes a service which broadcasts messages on many different subjects and a subscription request for only one or a few of those subjects. In other hypothetical examples, the service may publish only the requested information such as a particular telerate page. In the Telerate case, the subscription request may specify that only a particular page and particular columns of that page be sent and may request the page image by a point-to-point communication protocol using a dedicated TIB channel.

Step 434 represents the response processing of the service layer to the subscription request and the stream of data that results. In step 434, the service discipline does any necessary filtering by subject matter and assigns the TIB channel number. The filtering by subject matter is generally done by the service discipline on the data producer side of an exchange only when the producer produces vastly more data than is called for by the subscription such as in the case of a high speed, broadcast producer. In such a case, the extraneous data could overwhelm the network. The TIB channel numbers are assigned by the service discipline in step 434 but they are not actually added to the headers for the packets until the message reaches the communication layer. In some alternative embodiments, the TIB channel numbers may be written to the packet headers by the service discipline.

The TIB channel number assignment is done by the service discipline based upon the type of service, subscrip-

tion and communication protocol being used. Where a broadcast protocol is being used, the service discipline in some embodiments will, in step 434, simply assign different TIB channel numbers to different subjects and send a message to subscribers listed in a subscription table maintained by the service layer on the information layer. The message will say simply, for example, "For updates on IBM equity prices, monitor TIB channel 100". Note that the TIB channel data is used by the TIB software of the receiving host solely to route messages to the proper subscribing processes. TIB channels have nothing to do with logical channels, network routing or other network, data link or physical layer issues.

In other embodiments, in the broadcast protocol situation, the service discipline will consult a subscription list and filter out all messages on subjects other than subjects with open subscriptions. For those subjects, a TIB channels will be assigned and a message will be sent to the TIB software linked to the subscribing processes as to what TIB channels to listen to for messages to be routed to their client processes.

In the case of point-to-point protocols, the subscription requests usually contain the TIB channel numbers assigned to the subject by the service discipline selected by the information layer linked to the subscribing process. In such a case, step 434 represents the process of assigning the TIB channel number received in the subscription request to messages emitted from the service. In a typical case of a subscription to a Telerate page which specifies that a particular TIB channel is to be used in case a point-to-point protocol is selected, the service discipline will send the page image by selecting a point-to-point protocol engine. The service discipline will also send a message acknowledging the subscription and advising the TIB software of the subscribing process to listen to a particular TIB channel for broadcasts of updates to the page. The receiving TIB software then opens a TIB broadcast channel for the updates.

Step 436 represents the processes performed by the DCC library after the service discipline calls it. The DCC library's sole function in the preferred embodiment is to determine the best way to send a message to any particular network address where the service discipline or the subscription request does not specify the communication protocol to be used. In some embodiments, the DCC library of the communication layer will accept suggestions from the service layer or subscription request as to how to send the message but may select a different protocol if this is deemed to be more efficient.

Further, the DCC library may change the communication protocol being used based upon changing conditions such as number of subscribers. For example, an Intelligent Multicast protocol may be chosen (described in more detail below). In this protocol, a point-to-point protocol is used when the number of subscribers is below a certain cutoff number (programmable by the system administrator) but switchover to a broadcast protocol automatically occurs when the number of subscribers rises above the cutoff number. In the preferred embodiment "high-water" and "low-water" marks are used as will be described below. In other embodiments, any cost function may be used to set the switchover point based upon cost and efficiency of sending multiple point-to-point messages as opposed to a single broadcast message.

Step 436 also represents the process of retrieving the message from local memory of the service and putting it into an interprocess transfer process to send it to the protocol engine/daemon 30B in FIG. 15.

Step 438 represents the processes carried out by the protocol engine of the service to transmit the messages to the

subscribing processes. If the transport protocol in use is UDP, the protocol engine, in some embodiments, will do a packetizing function. This is the process of breaking down the message into packets and adding header data on the transmit side and reassembling the packets in the proper order on the receiver side. The TCP transport protocol does its own packetizing, so protocol engines that invoke this transport layer need not packetize. Nonpacket protocol engines also exist for other types of transport protocols.

The protocol engine also writes the port address of the machine running the subscribing process in the message headers and may perform other value added services. These other services include reliable broadcast and Intelligent Multicasting. Reliable broadcast services will be explained below, but basically this service provides functionality that does not exist in current broadcast communication protocols to increase reliability.

The protocol engines have a standard programmers interface through which they communicate with the transport layer routines in the operating system. The steps taken by the protocol engine to invoke the transport layer functionality so as to drive the network, data-link and physical layer protocols in such a manner so as to deliver the messages to the subscribing processes are not critical to the invention are symbolized by steps 440 and 442. Exactly what these steps are cannot be specified here because they are highly dependent upon the structure, configuration and protocol of the network as well as the interface to the transport layer. When any of these change, the protocol engines may have to be changed to accommodate the change to the network. This, however, prevents the need to change the application software thereby providing configuration decoupling.

After the message traverses the network, it is picked up by the network interface card having the port address shared by the subscribing process. This process is symbolized by step 444. The network card buffers the message and generates an interrupt to the transport layer routine which handles incoming messages.

Step 446 represents the process where the transport layer software calls the appropriate protocol engine of the daemon 30B in the communication layer such as layers 352 or 360 in FIG. 17. The incoming message or packet will be passed to the appropriate protocol engine by some interprocess transfer mechanism such as shared memory. In the preferred embodiment, the daemon is an ongoing process running in background on a multitasking machine. In other embodiments, the daemon is interrupt driven and only runs when a message has been received or is to be transmitted. Step 446 also represents the packet reassembly process for TCP or other transport layer protocols where packet reassembly is done by the transport layer.

Step 448 represents the processes performed by the protocol engine in the daemon to process and route the incoming message.

For UDP transport layer protocol engines, packet reassembly is done. This of course implies that the protocol engine of the data producer process added sequence numbers to the packet headers so that they can be reassembled in the proper order. Other value added services may then be performed such as checking all the sequence numbers against data which indicates the sequence numbers which should have arrived to determine if all the packets have been received. In some embodiments, the data as to the sequence numbers to expect is written into fields dedicated to this purpose in the packet headers. In other embodiments, this data is sent in a separate message.

If any packets are missing a message will be sent automatically by the receiving communication layer back to the

communication layer of the data producer process to request retransmission of any lost or garbled packets. This of course implies that the communication layer for the data process stores all packets in memory and retains them for possible retransmission until an acknowledgment message is received indicating that all packets have been successfully received.

Step 448 also symbolizes the main function performed by the communication layer daemon/protocol engine in receiving messages. That function is routing the messages to the appropriate subscribing process according to the TIB channel information in the header. The protocol engine checks the TIB channel number in the header against the current subscription list sent to it by the service discipline. The subscription list will include pointers to the appropriate service discipline callback routine and subscribing process for messages assigned to any particular TIB channel. The protocol engine also filters messages by TIB channel number for embodiments in which messages reach the TIB software coupled to the subscribing process which do not pertain to the subject of an open subscription. This may also be done at the service layer, or information layer but it is most efficient to do it at the communication layer.

The protocol engine will then put the message in the appropriate interprocess transfer mechanism, usually shared memory or a Unix™ pipe, and generate an interrupt to the DCC library as symbolized by step 450. This interrupt will vector processing to the appropriate DCC library callback routine which was identified to the protocol engine by the DCC library when the subscription on this TIB channel and subject was opened. The DCC library routine so invoked is linked to and part of the subscribing process which initiated the subscription. The DCC library callback routine then retrieves the message from the interprocess transfer mechanism and stores it in local memory of the subscribing process. The DCC library callback routine then generates an interrupt to the service layer and passes it a pointer to the message.

Step 452 represents the process performed by the service layer on incoming messages. The interrupt from the DCC library causes to run the service discipline callback routine identified in the original subscribe message passed by the service layer through the DCC library. The callback routine will, in some embodiments, do any data format conversions necessary and may, in other embodiments do subject matter filtering. Then, the service discipline generates an interrupt to the information layer which cause the callback routine of the information layer to run. The interrupt contains a pointer to the message.

Step 454 symbolizes processing of incoming messages by the information layer. In some embodiments, the service layer does not guarantee that all messages reaching the information layer exactly match the subject for which data was requested. In these embodiments, step 454 symbolizes the process of comparing the TIB channel code to the subject of the subscription to make sure they match. If the data has been previously filtered by subject, step 454 can be eliminated.

Step 456 symbolizes the process of generating an interrupt to the callback routine of the subscribing process if there is a match on subject. If not, no interrupt is generated and monitoring for new messages continues by the daemon while all the interrupt driven processes terminate and release their computer resources until the next interrupt.

Step 458 symbolizes the process of use of the message data for whatever purpose the subscribing process originally sought this data.

Reliable broadcast is one of the value added services that the communication layer can use to supplement and improve the communication protocols of the transport layer. Traditional broadcast protocols offered by prior art transport layers are not reliable. For example, if there is noise on the line which corrupts or destroys a packet or message or if the network interface card overflows the buffer, packets or entire messages can be lost and the processes listening for the message never gets the message, or they get an incomplete or garbled message. There is no acknowledge function in traditional broadcast, so if some of the processes miss the message or get incomplete or garbled messages, the transmitting process never finds out. This can happen for one in every hundred packets or for one in ten packet Traditional prior art transport layer broadcast protocols do not include functionality, i.e., program code, to distribute a broadcast message received at the network address of the host to multiple processes running on that host.

The communication layer according to the teachings of the invention includes at least one protocol engine to implement reliable broadcast protocols which are built on top and supplement the functionality of the prior art transport layer broadcast protocols. Referring to FIG. 21 comprised of FIGS. 21A and 21B, there is shown a flow chart for one embodiment of a reliable broadcast protocol implemented by the communication layer. Step 500 represents the process where the DCC library receives a request from a service discipline to send a message having a particular TIB channel assigned thereto. In some embodiments, this request may also include a request or a command to send the message by the reliable broadcast protocol. In embodiments where the reliable broadcast protocol is mandated by the service discipline, the service discipline includes a function to determine the number of subscribers to a particular channel and determine the cost of sending the same message many times to all the port addresses of all subscribers versus the cost of sending the message once by broadcast with messages to all subscribers to listen to TIB channel XX (whatever TIB channel number was assigned to this subject) for data on the subjects they are interested in. In the embodiment illustrated in FIG. 21, this cost determination function is included within the communication layer DCC library functionality.

Step 502 represents this cost determination process as performed by the DCC library. The particular program of the DCC library which implements this function, checks the subscription list and counts the number of subscribers to the TIB channel assigned to this message. The cost of sending the message point-to-point to all these subscribers is then evaluated using any desired costing function. In some embodiments, the cost function may be a comparison of the number of subscribers to a predetermined cutoff number. The particular cost function used is not critical to the invention. The cost of sending the message to multiple subscribers point-to-point is that the same message must be placed repeatedly on the network by the data producing software. The cost of broadcasting a message is that all network cards pick it up and may interrupt the transport protocol program in the operating system of the host which transmits the message by interprocess transfer to the TIB daemon only to find out the message is not of interest to any client process running on that host. Computer resources are thus wasted at any such host.

Step 504 represents the process the DCC library carries out to evaluate the cost and decide to send the message either by point-to-point protocol or reliable broadcast. If it is determined that the number of subscribers to this TIB channel is small enough, the decision will be made to send

the message by a point-to-point protocol. Step 506 represents the process of calling the point-to-point protocol engine and sending the message using this protocol.

If the number of subscribers is too high for efficient point-to-point transmission, the DCC library calls the reliable broadcast protocol engine as symbolized by step 508.

Step 510 represents the first step of the reliable broadcast protocol processing. The reliable broadcast protocol according to the teachings of the invention supports multiple subscribing processes running of the same host and requires that each subscribing process receive all the packets of the message without error and acknowledge receipt thereof. To insure that this is the case, sequence numbers must be added to the headers of each packet and some data must be communicated to the subscribing processes that indicate the sequence numbers that must all have been received in order to have received the entire message. In some embodiments, only the sequence numbers will be added to the packet headers and the data regarding the sequence numbers that comprise the entire message will be sent by separate message to each process having an open subscription to the TIB channel assigned to the message. In other embodiments, the sequence numbers that comprise the entire message will be added to the header of the first packet or to the headers of all the packets. The sequence numbers added to the packets are different than the sequence numbers added by packetizing functionality of the transport protocols of the operating system in TCP protocols since the TIB sequence numbers are used only to determine if all packets of a message have been received. In some embodiments, the packet sequence numbers added by the transport protocol may be used by the TIB communication layer of the subscribing processes to determine if all the packets have been received. In other embodiments of reliable broadcast protocol engines for supplementing the UDP transport layer protocol, the packetizing function of the protocol engine adds sequence numbers which can be used both for transport/network/data link/physical layer functions but also for TIB communication layer functions in verifying that all packets of a message have been received.

After the sequence numbers have been added, the packets are written to a retransmit buffer with their sequence numbers for storage in case some or all of the packets need to be retransmitted later as symbolized by step 512.

Before the messages can be sent to the various subscribing processes, the reliable broadcast protocol engine adds the TIB channel data to the header of each packet and sends a message to each subscribing process listed in the subscription table as having open subscriptions for this channel to listen for data on their requested subject on TIB channel XX where XX is the TIB channel number assigned to this subject.

Step 516 represents the process of transmitting the packets via the standard broadcast protocols of the transport layer by calling the appropriate operating system program and passing a pointer to each packet.

Referring to FIG. 22 comprised of FIGS. 22A and 22B, there is shown a flow chart of the processing by the communication layer of the subscribing process in the reliable broadcast protocol. The packets broadcast on the network are picked up by all network interface cards of all hosts on the network which then invoke the transport protocol software of the operating systems of the various hosts. The transport protocols then notify the daemons of the communication layers that a broadcast message has arrived and puts the packets in an interprocess transfer mechanism, usually shared memory. The daemons then retrieve the

packets from the interprocess transfer mechanism as represented by step 518.

Step 520 represents the process of checking the TIB channel numbers of the incoming packets to determine if they correspond to the TIB channel of any open subscription. If they do, the reliability sequence numbers are checked by the reliable broadcast protocol engine against the data indicating which packets and corresponding sequence numbers should have been received to have received a complete message. In some embodiments, especially embodiments using transport, network, data link and physical layers where error checking (ECC) is not performed at layers below the TIB interface software of the invention, error detection and correction is performed on the packets using the ECC bits appended to the packet. If errors have occurred that are beyond the range of correction given the number of ECC bits present, the packet is marked as garbled.

After determining which packets are missing or garbled, if any, the receiving protocol engine then sends a message back to the communication layer of the service or publishing process. This message will either acknowledge that all packets have been received without a problem or will request that certain packets be retransmitted. This is symbolized by step 522.

Step 524 represents the process of retransmission of the missing or garbled packets by the communication layer of the data producing process or service. In some embodiments, the missing or garbled packets will be sent point-to-point to only the subscribing process that did not get them. In other embodiments, the missing or garbled packets are broadcast to nodes with notification messages being sent to the subscribing processes that need them to listen on TIB channel XX where XX is the TIB channel on which the packets will be broadcast. The phrase "listen to channel XX" as it is used here has nothing to do with the actual transmission frequency, timeslot or other physical characteristic of the transmission. It merely means that the missing or garbled packets will be appearing on the network shortly and will have TIB channel XX routing information in their header data.

Step 526 represents the process of checking by the receiving communication layer that the replacement packets have been properly received similar to the processing of step 520. If they have, the receiving communication layer acknowledges this fact to the communication layer of the service. If not, a request for retransmission of the missing or garbled packets is again sent to the communication layer of the transmitting process, retransmission ensues and the whole process repeats until all packets have been successfully received. The final acknowledge message from the receiving communication layer to the transmitting communication layer that all packets have been successfully received causes the reliable broadcast protocol engine of the transmitting communication layer to flush all the packets from the retransmission memory as symbolized by step 528.

Step 530 represents the routing process where the reliable broadcast protocol engine checks the TIB channel data against the subscription list to determine which client processes have requested data assigned to this TIB channel. Once this information is known, the protocol engine passes a pointer to the message to all service disciplines which have entered subscriptions for data on this TIB channel. In some embodiments, the protocol engine will place a copy of the message in a separate interprocess transfer mechanism for every subscribing process. In other embodiments, shared memory will be the interprocess transfer mechanism and a pointer to the same copy of the message will be sent to all

subscribing processes. The subscribing processes will then arbitrate for access to the message in the information layer or the service layer.

Step 532 represents the processes previously described of passing the message up through the service and information layers to the subscribing process by successive interrupts causing to run the callback routines designated when the subscription was entered. Filtering by subject matter may also occur in some embodiments at the service layer and/or the information layer to guarantee a match to the subscribed subject.

FIG. 23 is a flow chart of processing to transmit data by the Intelligent Multicast communication protocol. This protocol uses either point-to-point or reliable broadcast protocols for each message depending upon the subject matter and how many subscriptions are open on this subject. The choice of protocol is automatically made for each message depending upon how many subscribing processes/network addresses there are for the message at the time the message is published. If the number of subscribers for a subject changes sufficiently, the transmission protocol may change automatically.

Step 600 represents the process of receiving a subscription request at the service layer of the data publishing process, passing this subscription along to the subscribing process and making an entry for a new subscription in the subscription table.

In step 602, a message is published by the service through the information layer to the service layer. The subject of the message may or may not be on the subject for which the new subscription was just entered. The service layer examines the subject data forwarded by the information layer about the message and coordinates with the information layer to assign a TIB channel to the subject if the TIB channel was already assigned by the service and information layers of the subscribing process as symbolized by step 604.

In step 606, the service discipline compares the number of subscribers for the subject of the message to a programmable cutoff number which is based upon the cost of transmission point-to-point versus the cost of transmission by reliable broadcast. The programmable cutoff number can be set and altered by the system administrator and is based upon any desired cost function, the nature of which is not critical to the invention. In the preferred embodiment, the cost function is comprised of a high water mark and a low water mark. If the number of subscribers is above the high water mark, the message will be sent by reliable broadcast. If the number of subscribers to this subject then subsequently falls below the low water mark, subsequent messages will be sent point-to-point. In some embodiments, the cost function can be an automatic learning program that listens to the network and subscription requests and makes the decision based upon latency time or some other criteria of network efficiency.

Step 608 represents the process of calling the reliable broadcast protocol engine if the number of subscribers is greater than the cutoff number. The message is then put in an interprocess transfer mechanism directed to this protocol engine.

If the number of subscribers is below the cutoff number, point-to-point transmission is more efficient so the service discipline calls the point-to-point protocol engine and puts the message into an interprocess transfer mechanism directed to this protocol engine as symbolized by step 610.

Step 612 represents the process of waiting for the next message or subscription and returning to step 600 if a subscription is received and to step 602 if another message is received.

In summary the concept of the invention is to use software layers to decouple applications from the complexities of the computer network communication art in ways that applications have never before been decoupled. For example, it is believed that the subject based addressing decoupling provided by the information layer is new especially when coupled with the service decoupling provided by the service layer. It is believed to be new to have extensible service and communication layers that can be easily modified by the addition of new service disciplines and protocol engines to provide service and configuration decoupling under changing conditions such as new network topologies and the addition of new or changed services and/or servers to the network. It is new to have a service layer that includes many different service disciplines designed to encapsulate many varied communication protocols. For example, these service disciplines can handle communication with everything from services like Telnet to operating system programs, other processes on other machines (or even the same machine or another part of the same process even) to a user sitting at a terminal. Further, the abilities of this service layer to implement failure monitoring and recovery, distribution and replication management, and security/access control services is new.

Further, it is new to have the configuration decoupling and value added services of the communication layer.

The teachings of the invention contemplate use of any one of these layers or any combination of the three in the various embodiments which together define a class or genus of software programs the species of which implement the specific functions or combinations thereof defined herein.

Appendix A, which can be found in the application file, is a complete source code listing for the TIB™ communication interface software according to the teachings of the invention in the C programming language. Included are all library programs, all interfaces and all layers of the software for both data publishing and data consuming processes. Also included are all utility programs necessary to compile this software into machine readable code for a Unix™ based multitasking workstation.

There follows a more detailed specification of the various library programs and the overall structure and functioning of an embodiment of the communication interface according to the teachings of the invention.

Information Driven Architecture™, Teknekron Information Bus™, TIB™, TIBINFO™, TIBFORMS™, Subject-Based Addressing™, and RMDP™ are trademarks of Teknekron Software Systems, Inc.

CONTENTS

1. Introduction
2. Teknekron Information Bus Architecture
3. Reliable Market Data Protocol:RMDP
4. Subject-Addressed Subscription Service:TIBINFO
5. Data-exchange Component:TIBFORM
6. Appendix: 'man' Pages

1. Introduction

The Teknekron Information Bus™ software (TIB™ component) is a distributed software component designed to facilitate the exchange of data among applications executing in a real-time, distributed environment. It is built on top of industry standard communication protocols (TCP/IP) and data-exchange standards (e.g., X.400).

The document is organized as follows. Section 2 gives an architectural overview of the TIB™. Section 3 describes the

Reliable Market Data Protocol. This general purpose protocol is particularly well suited to the requirements of the page-based market data services. It is also often used for bulletin and report distribution. Section 4 describes TIB-INFO, an interface supporting Subject-based Addressing. Section 5 describes a component and its interface that supports a very flexible and extensible data-exchange standard. This component is called TIBFORMS. The Appendix contains (UNIX-like) manual pages for the core interfaces.

2. Architectural Overview

2.1 Introduction

The Teknekron Information Bus (TIB™) is comprised of two major components: the (application-oriented) data communication component and the data-exchange component. These are depicted in FIG. 2.1. In addition, a set of presentation tools and a set of support utilities have been built around these components to assist the application developer in the writing of TIB™-based applications.

The (application-oriented) data communication component implements an extensible framework for implementing high-level, communication protocol suites. Two protocol suites have been implemented that are tailored toward the needs of fault-tolerant, real-time applications that communicate via messages. Specifically, the suites implement subscription services that provide communication support for monitoring dynamically changing values over a network. Subscription services implement a communication paradigm well suited to distributing market data from, for example, Quotron or Telerate.

One of the protocol suites supports a traditional service-oriented cooperative processing model. The other protocol suite directly supports a novel information-oriented, cooperative processing model by implementing subject-based addressing. Using this addressing scheme, applications can request information by subject through a general purpose interface. Subject-based addressing allowing information consumers to be decoupled from information producers; thereby, increasing the modularity and extensibility of the system.

The application-oriented protocol suites are built on top of a common set of communication facilities called the distributed communications component, depicted as a sublayer in FIG. 2.1. In addition to providing reliable communications protocols, this layer provides location transparency and network independence to its clients.

The layer is built on top of standard transport-layer protocols (e.g., TCP/IP) and is capable of supporting multiple transport protocols. The data-exchange component implements a powerful way of representing and transmitting data. All data is encapsulated within self-describing data objects, called TIB™-forms or, more commonly, simply forms. Since TIB™ forms are self-describing, they admit the implementation of generic tools for data manipulation and display. Such tools include communication tools for sending forms between processes in a machine-independent format. Since a self-describing form can be extended without adversely impacting the applications using it, forms greatly facilitate modular application development.

The two major components of TIB™ were designed so that applications programmers can use them independently or together. For example, forms are not only useful for communicating applications that share data, but also for non-communicating applications that desire to use the generic tools and modular programming techniques supported by forms. Such applications, of course, do not need the communication services of the TIB™. Similarly, appli-

cations using subject-based addressing, for example, need not transmit forms, but instead can transmit any data structure. Note that the implementation of the communication component does use forms, but it does not require applications to use them.

2.2 System Model

The system model supported by the TIB™ consists of users, user groups, networks, services, service instances (or servers), and subjects.

The concept of a user, representing a human "end-user," is common to most systems. A user is identified by a user-id. The TIB™ user-id is normally the same as the user-id (or logon id) supported by the underlying operating system, but it need not be.

Each user is a member of a exactly one group. The intention is that group should be composed of users with similar service access patterns and access rights. Access rights to a service or system object are grantable at the level of users and at the level of groups. The system administrator is responsible for assigning users to groups.

A network is a logical concept defined by the underlying transport layer and is supported by the TIB™. An application can send or receive across any of the networks that its host machine is attached to. It also supports all gateways functions and internetwork routing that is supported by the underlying transport-layer protocols.

Since the lowest layer of the TIB™ communication component supports multiple networks, application-oriented protocols can be written that transparently switchover from one network to another in the event of a network failure.

A service represents a meaningful set of functions that are exported by an application for use by its clients. Examples of services are an historical news retrieval service, a Quotron datafeed, and a trade ticket router. An application will typically export only one service, although it can export many different services.

A service instance is an application process capable of providing the given service. (Sometimes these are called "server processes.") For a given service, several instances may be concurrently providing it, so as to improve performance or to provide fault tolerance. Application-oriented communication protocols in the TIB™ can implement the notion of a "fault-tolerant" service by providing automatic switchover from a failed service instance to an operational one providing the same service.

Networks, services, and servers are traditional components of a system model and are implemented in one fashion or another in most distributed systems. On the other hand, the notion of a subject is novel to the information model implemented by the TIB™.

The subject space consists of a hierarchical set of subject categories. The current release of the TIB™ supports a 4 level hierarchy, as illustrated by the following well formed subject: "equity.ibm.composite.trade." The TIB™ itself enforces no policy as to the interpretation of the various subject categories. Instead, the applications have the freedom and responsibility to establish conventions on use and interpretation of subject categories.

Each subject is typically associated with one or more services producing data about that subject. The subject-based protocol suites of the TIB™ are responsible for translating an application's request for data on a subject into communication connections to one or more service instances providing information on that subject.

A set of subject categories is referred to as a subject domain. The TIB™ provides support for multiple subject

domains. This facility is useful, for example, when migrating from one domain to another domain. Each domain can define domain-specific subject encoding functions for efficiently representing subjects in message headers.

2.3 Process Architecture

The communication component of the TIB™ is a truly distributed system with its functions being split between a frontend TIB™/communication library, which is linked with each application, and a backend TIB™/communication daemon process, for which there is typically one per host processor. This process architecture is depicted FIG. 2.2. Note that this functional split between TIB™ library and TIB™ daemon is completely transparent to the application. In fact, the application is completely unaware of the existence of the TIB™ daemon, with the exception of certain failure return codes.

The TIB™ daemons cooperate among themselves to ensure reliable, efficient communication between machines. For subject-addressed data, they assist in its efficient transmission by providing low-level system support for filtering messages by subject.

The TIB™/communication library performs numerous functions associated with each of the application-oriented communication suites. For example, the library translates subjects into efficient message headers that are more compact and easier to check than ASCII subject values. It also maps service requests into requests targeted for particular service instances, and monitors the status of those instances.

The data-exchange component of TIB™ is implemented as a library, called the TIB™/form library, that is linked with the application. This library provides all of the core functions of the data-exchange component and can be linked independently of the TIB™/communication library. The TIB™/form library does not require the TIB™/communication daemon.

2.4 Communication Component

The TIB™ Communication Component consists of 3 subcomponents: the lower-level distributed communication component (DCC), and two high-level application-oriented communication protocol suites—the Market Data Subscription Service (MDSS), and the Subject-Addressed Subscription Service (SASS).

The high-level protocol suites are tailored around a communication paradigm known as a subscription. In this paradigm, a data consumer “subscribes” to a service or subject, and in return receives a continuous stream of data about the service or subject until the consumer explicitly terminates the subscription (or a failure occurs). A subscription paradigm is well suited for realtime applications that monitor dynamically changing values, such as a stock’s price. In contrast, the more traditional request/reply communication paradigm is ill-suited for such realtime applications, since it requires data consumers to “poll” data providers to learn of changes.

The principal difference between the two high-level protocols is that the MDSS is service-oriented and SASS is subject-oriented. Hence, for example, MDSS supports the sending of operations and messages to services, in addition to supporting subscriptions; whereas, SASS supports no similar functionality.

2.4.3. Market Data Subscription Service

2.4.3.1 Overview

MDSS allows data consumers to receive a continuous stream of data, tolerant of failures of individual data sources. This protocol suite provides mechanisms for administering load balancing and entitlement policies.

Two properties distinguish the MDSS protocols from the typical client/server protocols (e.g. RPC). First, subscriptions are explicitly supported, whereby changes to requested values are automatically propagated to clients. Second, clients request (or subscribe) to a service, as opposed to a server, and it is the responsibility of the MDSS component to forward the client’s request to an available server. The MDSS is then responsible for monitoring the server connection and reestablishing it if fails, using a different server, if necessary.

The MDSS has been designed to meet the following important objectives:

(1) Fault tolerance. By supporting automatic switchover between redundant services, by explicitly supporting dual (or triple) networks, and by utilizing the fault-tolerant transmission protocols implemented in the DCC (such as the “reliable broadcast protocols”), the MDSS ensures the integrity of a subscription against all single point failures. An inopportune failure may temporarily disrupt a subscription, but the MDSS is designed to detect failures in a timely fashion and to quickly search for an alternative communication path and/or server. Recovery is automatic as well.

(2) Load balancing. The MDSS attempts to balance the load across all operational servers for a service. It also rebalances the load when a server fails or recovers. In addition, the MDSS supports server assignment policies that attempts to optimize the utilization of scarce resources such as “slots” in a page cache or bandwidth across an external communication line.

(3) Network efficiency. The MDSS supports the intelligent multicast protocol implemented in the DCC. This protocol attempts to optimize the limited resources of both network bandwidth and processor I/O bandwidth by providing automatic, dynamic switchover from point-to-point communication protocols to broadcast protocols. For example, the protocol may provide point-to-point distribution of Telerate page 8 to the first five subscribers and then switch all subscribers to broadcast distribution when the sixth subscriber appears.

(4) High-level communication interface. The MDSS implements a simple, easy-to-use application development interface that mask most of the complexities of programming a distributed system, including locating servers, establishing communication connections, reacting to failures and recoveries, and load balancing.

2.4.1.2 Functionality

The MDSS supports the following core functions:

get MDSS establishes a fault-tolerant connection to a server for the specified service and “gets” (i.e., retrieves) the current value of the specified page or data element. The connection is subscription based so that updates to the specified page are automatically forwarded.

halt “halt” the subscription to the specified service.

derive sends a modifier to the server that could potentially change the subscription.

The MDSS protocol has been high-optimized to support page-oriented market data feed, and this focus has been reflected in the choice of function names. However, the protocol suite itself is quite general and supports the distribution of any type of data. Consequently, the protocol suite is useful and is being used in other contexts (e.g., data distribution in an electronic billboard). 2.4.2 Subject-Addressed Subscription Service (SASS) 2.4.2.1 Overview

The SASS is a sophisticated protocol suite providing application developers a very high-level communications

interface that fully supports the information-oriented, cooperative processing model. This is achieved through the use of subject-based addressing.

The basic idea behind subject-based addressing and the SASS's implementation of it is straightforward. Whenever an application requires a piece of data, especially, data that represents a dynamically changing value (e.g. a stock price), the application simply subscribes to that data by specifying the appropriate subject. For example, in order to receive all trade tickets on IBM, an application may issue the following subscription: "trade_ticket.IBM". Once an application has subscribed to a particular subject, it is the responsibility of the SASS to choose one or more service instances providing information on that subject. The SASS then makes the appropriate communications connections and (optionally) notifies the service instances providing the information.

The SASS has been designed to meet several important objectives:

(1) Decoupling information consumers from information providers. Through the use of subject-based addressing, information consumers can request information in a way that is independent of the application producing the information. Hence, the producing application can be modified or supplanted by a new application providing the same information without affecting the consumers of the information.

(2) Efficiency. Support for filtering messages by subject is built into the low levels of the TIB™ daemon, where it can be very efficient. Also, the SASS supports filtering data at the producer side: data that is not currently of interest to any application can simply be discarded prior to placing in on the network; thereby, conserving network bandwidth and processor I/O bandwidth.

(3) High-level communication interface. The SASS interface greatly reduces the complexities of programming a distributed application in three ways. First, the consumer requests information by subject, as opposed to by server or service. Specifying information at this level is easier and more natural than at the service level. Also, it insulates the program from changes in service providers (e.g., a switch from IDN to Ticker 3 for equity prices). Second, the SASS presents all data through a simple uniform interface—a programmer needing information supplied by three services need not learn three service-specific protocols, as he would in a traditional processing model. Third, the SASS automates many of the hard or error-prone tasks, such as searching for an appropriate service instance, and establishing the correct communication connection.

2.4.2.2 Functionality

For a data consumer, the SASS provides three basic functions:

subscribe where the consumer requests information on a real-time basis on one or more subjects. The SASS components sets up any necessary communication connections to ensure that all data matching the given subject(s) will be delivered to the consumer. The consumer can specify that data be delivered either asynchronously (interrupt-driven) or synchronously. A subscription may result in the producer service instance being informed of the subscription. This occurs whenever the producer has set up a registration procedure for its service. This notification of the producer via any specified registration procedure is transparent to the consumer.

cancel which is the opposite of subscribe. The SASS component gracefully closes down any dedicated communication channels, and notifies the producer if an appropriate registration procedure exists for the service.

receive receive and "callbacks" are two different ways for applications to receive messages matching their subscriptions. Callbacks are asynchronous and support the event driven programming style—a style that is particularly well-suited for applications requiring realtime data exchange. "Receive" supports a traditional synchronous interface for message receipt.

For a data producer, the SASS provides a complementary set of functions.

Note that an application can be both a producer and a consumer with respect to the SASS, and this is not uncommon.

2.4.3 Distributed Communication Component

2.4.3.1 Overview

The Distributed Communication Component (DCC) provides communication services to higher-level TIB™ protocols, in particular, it provides several types of fault transparent protocols.

The DCC is based on several important objectives:

(1) The provision of a simple, stable, and uniform communication model. This objective offers several benefits. First, it offers increased programmer productivity by shielding developers from the complexities of a distributed environment; locating a target process, establishing communications with it, and determining when something has gone awry are all tasks best done by a capable communications infrastructure, not by the programmer. Second, it reduces development time, not only by increasing programmer productivity, but also by simplifying the integration of new features. Finally, it enhances configurability by keeping applications unaware of the physical distribution of other components. This prevents developers from building in dependencies based on a particular physical configuration. (Such dependencies would complicate subsequent reconfigurations.)

(2) Portability through encapsulation of important system structures. This objective achieves importance when migration to a new hardware or software environment becomes necessary. The effort expended in shielding applications from the specific underlying communication protocols and access methods pays off handsomely at that time. By isolating the required changes in a small portion of the system (in this case, the DCC), applications can be ported virtually unchanged, and the firm's application investment is protected.

(3) Efficiency. This is particularly important in this component. To achieve this, the DCC builds on top of less costly "connectionless" transport protocols in standard protocol suites (e.g., TCP/IP and OSI). Also, the DCC has been carefully designed to avoid the most costly problem in protocols: the proliferation of data "copy" operations.

The DCC achieves these objectives by implementing a layer of services on top of the basic services provided by vendor-supplied software. Rather than re-inventing basic functions like reliable data transfer or flow-control mechanisms, the DCC concentrates on shielding applications from the idiosyncrasies of any one particular operating system. Examples include the hardware-oriented interfaces of the MS-DOS environment, or the per-process file descriptor limit of UNIX. By providing a single, unified communication tool that can be easily replicated in many hardware or software environment, the DCC fulfills the above objectives.

2.4.3.2 Functionality

The DCC implements several different transmission protocols to support the various interaction paradigms, fault-tolerance requirements, and performance requirements imposed by the high-level protocols. Two of the more

interesting protocols are reliable broadcast and intelligent multicast protocols.

Standard broadcast protocols are not reliable and are unable to detect lost messages. The DCC reliable broadcast protocols ensure that all operational hosts either receive each broadcast message or detects the loss of the message. Unlike many so-called reliable broadcast protocols, lost messages are retransmitted on a limited, periodic basis.

The intelligent multicast protocol provides a reliable datastream to multiple destinations. The novel aspect of the protocol is that it can dynamically switch from point-to-point transmission to broadcast transmission in order to optimize the network and processor load. The switch from point-to-point to broadcast (and vice versa) is transparent to higher-level protocols. This protocol admits the support of a much larger number of consumers than would be possible using either point-to-point or broadcast alone. The protocol is built on top of other protocols within the DCC.

Currently, all DCC protocols exchange data only in discrete units, i.e., "messages" (in contrast to many Transport protocols). The DCC guarantees that the messages originating from a single process are received in the order sent.

The DCC contains fault-tolerant message transmission protocols that support retransmission in the event of a lost message. The package guarantees "at-most-once" semantics with regards to message delivery and makes a best attempt to ensure "exactly once" semantics.

The DCC contains no exposed interfaces for use by application developers.

3. RELIABLE MARKET DATA PROTOCOL

3.1 Introduction

The Reliable Market Data Protocol (RMDP) defines a programmatic interface to the protocol suite and services comprising the Market Data Subscription Service (MDSS) TIB™ subcomponent. RMDP allows market data consumers to receive a continuous stream of data, based on a subscription request to a given service. RMDP tolerates failures of individual servers, by providing facilities to automatically reconnect to alternative servers providing the same service. All the mechanisms for detecting server failure and recovery, and for hunting for available servers are implemented in the RMDP library. Consequently, application programs can be written in a simple and naive way.

The protocol provides mechanisms for administering load balancing and entitlement policies. For example, consider a trading room with three Telerate lines. To maximize utilization of the available bandwidth of those Telerate lines, the system administrator can "assign" certain commonly used pages to particular servers, i.e., page 5 to server A, page 405 to server B, etc. Each user (or user group) would be assigned a "default" server for pages which are not explicitly preassigned. (These assignments are recorded in the TIB™ Services Directory.)

To accommodate failures, pages or users are actually assigned to prioritized list of servers. When a server experiences a hardware or software failure, RMDP hunts for and connects to the next server on the list. When a server recovers, it announces its presence to all RMDP clients, and RMDP reconnects the server's original clients to it. (Automatic reconnection avoids situations where some servers are overloaded while others are idle.) Except for status messages, failure and recovery reconnections are transparent to the application.

The MDSS protocol suite, including RMDP, is built on top of the DCC and utilizes the reliable communication protocols implemented in that component. In particular, the MDSS suite utilizes the reliable broadcast protocols and the

intelligent multicast protocol provided therein. RMDP supports both LANs and wide area networks (WANs). RMDP also supports dual (or multiple) networks in a transparent fashion.

RMDP is a "service-addressed" protocol; a complementary protocol, TIBINFO, supports "subject-based addressing."

3.2 Programmatic Interface

RMDP programs are event-driven. All RMDP function calls are non-blocking: even if the call results in communication with a server, the call returns immediately. Server responses, as well as error messages, are returned at a later time through an application-supplied callback procedure.

The principal object abstraction implemented in RMDP is that of an Rstream, a "reliable stream," of data that is associated with a particular subscription to a specified service. Although, due to failures and recoveries, different servers may provide the subscription data at different times, the Rstream implements the abstraction of a single unified data stream. Except for short periods during failure or recovery reconnection, an Rstream is connected to exactly one server for the specified service. An application may open as many Rstreams as needed, subject only to available memory.

An Rstream is bidirectional—in particular, the RMDP client can send control commands and messages to the connected server over the Rstream. These commands and messages may spur responses or error messages from the server, and in one case, a command causes a "derived" subscription to be generated. Regardless of cause, all data and error messages (whether remotely or locally generated) are delivered to the client via the appropriate Rstream.

The RMDP interface is a narrow interface consisting of just six functions, which are described below.

```
void
rmdp_SetProp(property, value)
rmdp_prop_t property;
caddr_t value;
```

Used to set the values of RMDP properties. These calls must be made before the call to `rmdp_Init()`. Required properties are marked with ? in the list below. Other properties are optional. The properties currently used are:

*RMDP_CALLBACK

Pointer to the callback function. See the description of callback below.

RMDP_SERVICE_MAP

The name of Services Directory to be used in lieu of the standard directory.

RMDP_GROUP

The user group used to determine the appropriate server list. Should be prefixed with '+'. Default is group is "+" (i.e. the null group).

RMDP_RETRY_TIME

The number of seconds that the client will wait between successive retries to the same server, e.g., in the case of cache full." Default is 30.

57

RMDP_QUIET_TIME

The time in seconds that a stream may be "quiet" before the protocol assumes that the server has died and initiates a "hunt" for a different server. Default is 75.

RMDP_VERIFY_TIME

The time in seconds between successive pings of the server by the client. Default is 60.

RMDP_APP_NAME

The name of the application i.e. "telerate", "reuters" etc. If this property is set, then the relevant entries from the Service Directory will be cached.

```
void
rmdp_init( );
```

This initializes the internal data structures and must be called prior to any calls to rmdp_Get().

```
RStream
rmdp_Get(service, request, host)
char *service, *request, *host;
```

This is used to get a stream of data for a particular 'service' and subscription 'request'. For the standard market data services, the request will be the name of a page (e.g., "5", "AANN"). If 'host' is non-NULL, then the RMDP will only use the server on the given host. In this case, no reconnection to alternative servers will be attempted upon a server failure. If 'host' is NULL, then RMDP will consult the TIB™ Services Directory to identify a list of server alternatives for the request. 'rstream' is an opaque value that is used to refer to the stream. All data passed to the application's callback function will be identified by this value.

An error is indicated by RStream rstream = NULL.

```
RStream
rmdp_Derive(rstream, op)
RStreamold;
char *op;
```

This generates a new subscription and, hence, a new 'Rstream' from an existing subscription. 'command' is a string sent to the server, where it is interpreted to determine the specific derivation.

The standard market data servers understand the following commands: "n" for next-page, "p" for previous-page and "t XXXX" for time-page.

Derived streams cannot be recovered in the case of server failure. If successful, an Rstream is returned, otherwise NULL is returned.

```
void
rmdp_Message(rstream, msg)
RStreamrstream;
char *msg;
```

Sends the string 'msg' to the server used by 'rstream'. The messages are passed directly to the server, and are not in any way affected by the state of the stream. The messages are understood by the standard market data servers include "rr <PAGE NAME>" to rerequest a page, and "q a" to request

58

the server's network address. Some messages induce a response from the server (such as queries). In this case, the response will be delivered to all streams that are connected to the server.

```
void
rmdp_Halt(rstream)
RStreamrstream;
```

This gracefully halts the 'rstream'.

```
void
callback(rstream, msgtype, msg, act, err)
RStreamrstream;
mdp_msg_t      msgtype;
char           *msg;
mdp_act_t      act;
mdp_err_t      err;
```

This is the callback function which was registered with rmdp_SetProp(RMDP_CALLBACK, callback). 'rstream' is the stream to which the message pertains. 'msgtype' can be any of the values defined below (see "RMDP Message Type"). 'msg' is a string which may contain vt100 compatible escape sequences, as in MDSS. (It will NOT however be prefaced with an "[... E. That role is assumed by the parameter 'msgtype'.)

The last two parameters are only meaningful if 'msgtype' is MDP_MSG_STATUS. 'act' can be any of the values found in "RMDP Action Type" (see below), but special action is necessary only if act = 'MDP_ACT_CANCEL'. The latter indicates that the stream is being canceled and is no longer valid. It is up to the application to take appropriate action. In either case, 'err' can be any of the values found in "RMDP Error Type" (see below), and provides a description of the status.

RMDP Message Types (mdp_msg_t)

The message types are listed below. These types are defined in the underlying (unreliable) Market Data Protocol (MDP) and are exported to the RMDP.

```
MDP_MSG_BAD = -1      Page data message.
MDP_MSG_DATA = 0      Status/error message.
MDP_MSG_STATUS = 1    "Out of Band" message, e.g.,
MDP_MSG_OOB = 2       time stamp.
MDP_MSG_QUERY = 3     Query result.
```

RMDP Action Type (mdp_act_t)

The action types are listed below. These action types inform the RMDP clients of activities occurring in the lower level protocols. Generally speaking, they are "for your information only" messages, and do not require additional actions by the RMDP client. The exception is the "MDP_ACT_CANCEL" action, for which there is no recovery. These types are defined in the underlying (unreliable) Market Data Protocol (MDP) and are exported to the RMDP.

```
MDP_ACT_OK = 0        No unusual action required.
MDP_ACT_CANCEL = 1    The request cannot be
                      serviced, cancel the stream,
                      do not attempt to reconnect
                      (E.g., invalid page name.)
```


MDP_ACT_CONN_FIRST = 2	The server is closing the stream; the first server in the alternatives list is being tried. (E.g., the server is shedding "extra" clients for load balancing.)
MDP_ACT_CONN_NEXT = 3	The server is closing the stream; the next server in the alternatives list is being tried. (E.g., the server's line to host fails.)
MDP_ACT_LATER = 4	Server cannot service request at this time; will re-submit request later, or try a different server. (E.g., Cache full.)
MDP_ACT_RETRY = 5	Request is being retried immediately.

RMDP Error Types (mdp_err_t)

Description of error, for logging or reporting to end user. These types are defined in the underlying (unreliable) Market Data Protocol (MDP) and are exported to the RMDP.

```

MDP_ERR_OK = 0
MDP_ERR_LOW = 1
MDP_ERR_QUIET = 2
MDP_ERR_INVALID = 3
MDP_ERR_RESRC = 4
MDP_ERR_INTERNAL = 5
MDP_ERR_DELAY = 6
MDP_ERR_SYS = 7
MDP_ERR_COMM = 8

```

4. Subject-Addressed Subscription Service: TIBINFO

4.1 Introduction

TIBINFO defines a programmatic interface to the protocols and services comprising the TIB™ subcomponent providing Subject-Addressed Subscription Services (SASS). The TIBINFO interface consists of libraries: TIBINFO_CONSUME for data consumers, and TIBINFO_PUBLISH for data providers. An application includes one library or the other or both depending on whether it is a consumer or provider or both. An application can simultaneously be a consumer and a producer.

Through its support of Subject-Based Addressing, TIBINFO supports a information-oriented model of cooperative processing by providing a method for consumers to request information in a way that is independent of the service (or services) producing the information. Consequently, services can be modified or replaced by alternate services providing equivalent information without impacting the information consumers. This decoupling of information consumers from information providers permits a higher degree of modularization and flexibility than that permitted by traditional service-oriented processing models.

For Subject-Based Addressing to be useful in a real time environment, it must be efficiently implemented. With this objective in mind, support for Subject-Based Addressing has been built into the low levels of the Distributed Communications Component. In particular, the filtering of messages by subject is performed within the TIB™ daemon itself.

4.2 Concepts

Subject

The subject space is hierarchical. Currently, a 4-level hierarchy is supported of the following format:

```
major[.minor[.qualifier1[.qualifier2]]]
```

where '[' and ']' are metacharacters that delimit an optional component. major, minor, qualifier1 and qualifier2 are called subject identifiers. A subject identifier is a string consisting of the printable ascii characters excluding '.', '?', and '*'. A subject identifier can be an empty string, in which case it will match with any subject identifier in that position. The complete subject, including the '.' separators, cannot exceed 32 characters. Subjects are case sensitive.

Some example of valid subjects are listed below: The comments refer to the interpretation of subjects on the consume side. (The publish-side semantics are slightly different.)

equity.ibm.composite.quote	
equity.composite.quote	matches any minor subject
equity.ibm	matches any qualifier1 and qualifier2
equity.ibm.	same as above

Within the TIBINFO and the SASS, subjects are not interpreted. Hence, applications are free to establish conventions on the subject space. It should be noted that SASS components first attempt to match the major and minor subject identifiers first. As a consequence, although applications can establish the convention that "equity.ibm" and ".equity.ibm" are equivalent subjects, subscriptions to "equity.ibm" will be more efficiently processed.

Stream

A stream is an abstraction for grouping subscriptions. The subscriptions on a stream share a common set of properties, notably the same message handler (i.e., "callback" routine) and the same error handler. All subscriptions on a stream can be "canceled" simply by destroying the stream.

A stream imposes little overhead on the system. They can therefore be freely created and destroyed.

Protocol Engines, Service Disciplines, and Subject Mappers

The SASS and DCC components implement many support services in order to provide the functionality in TIBINFO. These include subject mappers for efficiently handling subjects, service disciplines for controlling the interaction with servers, and protocol engines for implementing reliable communication protocols. TIBINFO provides an interface for setting properties of these components. Hence, by setting the appropriate properties, one can specify, for example, the behavior of the subject mapper through the TIBINFO interface. Since these properties are in configuration files, configuration and site dependent parameters can be altered for the above components by the system administrator through TIBINFO.

In some embodiments, the property definitions for TIBINFO and for the underlying components may be augmented to support enhancements. This use of properties yields flexibility and extensibility within the confines of a stable functional interface.

4.3 Description

The TIBINFO interface is high-level and easy to use. Published data can be a form or an uninterpreted byte string. Messages can be received either in a synchronous fashion,

61

or in an asynchronous fashion that is suitable for event-driven programming. The following functions are sufficient to write sophisticated consumers using event-driven programming.

```
Tib_stream *tib_consume_create(property-list,
                               TIB_EOP)
```

Creates a TIBINFO stream that supports multiple subscriptions via the "subscribe" function. The property_list is a (possibly empty) list of property value pairs, as illustrated by

```
tib_consume_create(TIB_PROP_MSGHANDLER,
                  my_handler,
                  TIB_PROP_ERRHANDLER, my_err_handler, TIB_EOP);
```

Valid properties are defined below. TIB_EOP is a literal signaling the end of the property list.

```
void tib_destroy(stream)
Tib_stream *stream;
```

Reclaims resources used by the specified stream.

```
Tib_errorcode tib_subscribe(stream, subject, clientdata)
Tib_stream *stream;
Tib_subject *subject;
caddr_t clientdata;
```

Notifies the TIB™ software that the client application is interested in messages having the indicated subject. If stream has an associated "message-handler," then it will be called whenever a message satisfying the subscription arrives. Qualifying messages are delivered on a first-in/first-out basis. The value of clientdata is returned in every message satisfying the subscription subject. Note that multiple subscriptions to the same subject on the same stream are undefined.

```
void tib_cancel(stream)
Tib_stream *stream;
```

Cancels the client application's subscription to the specified subject.

```
void my_message_handler(stream,msg)
Tib_stream *stream;
Tib_message *message;
```

This is the "callback" function that was registered with the stream. Forms are returned unpacked. The function can reference the entire message structure through the macros described below.

62

The following functions are sufficient to write producers. Two publishing functions are provided to support the different data types that can be transmitted through the TIB-INFO interface.

```
tib_publish_create(property-list, TIB_EOP)
```

Is used to create an TIBINFO stream for publishing records. The property_list is a (possibly empty) list of property-value pairs, as illustrated by

```
tib_publish_create(TIB_PROP_ERRHANDLER,my_handler,TIB_EOP)
```

Valid properties are defined below. TIB_EOP is a constant signaling the end of the property list.

```
tib_destroy(stream)
Tib_stream stream;
```

Reclaims resources used by the specified stream.

```
Tib_errorcode tib_publish_form(stream, subject, form)
Tib_stream *stream;
Tib_subject *subject;
Form form;
```

Accepts a single, unpacked form, packs it, and publishes it.

```
Tib_errorcode tib_publish_buffer(stream, subject, length,
form)
Tib_stream *stream;
Tib_subject *subject;
short length;
Form form;
```

Accepts a byte buffer of specified length and publishes it.

The remaining functions are control functions that apply to both the consume and the publish side.

```
void Tib_batch()
```

This may be used prior to initiating multiple subscriptions. It informs the TIB™ library that it can delay acting on the subscriptions until a tib_unbatch is seen. This allows the TIB™ library to attempt to optimize the execution of requests. Note that no guarantees are made about the ordering or timing of "hatched" request. In particular, (i) requests may be executed prior to the receipt of the tib_unbatch function, and (ii) the effects of changing properties in the middle of a hatched sequence of requests is undefined. Batch and unbatch requests may be nested. (Note that the use of tib_batch is completely optional and it does not change the semantics of a correct program.)

```
Tib_errorcode tib_stream_set(stream, property, value)
Tib_stream *stream;
```

```
Tib_property  *property;
caddr_t      value;
```

Used to change the dynamically settable properties of a stream. These properties are described below. Note that some properties can only be set prior to stream creation (via `tib_default_set`) or at stream creation.

```
caddr_t tib_stream_get(stream, property)
Tib_stream *stream;
Tib_property *property;
```

Used to retrieve the current value of the specified property.

```
Tib_errcode tib_default_set( property, value)
Tib_stream *stream;
Tib_property *property;
caddr_t      value;
```

Used to change the initial properties of a stream. During stream creation, the default values are used as initial values in the new stream whenever a property value is not explicitly specified in the creation argument list.

```
Tib_errcode tib_default_get( property)
Tib_stream *stream;
Tib_property *property;
```

Used to retrieve the default value of the specified property.

`tib_unbatch()`

Informs TIBINFO to stop "batching" functions and to execute any outstanding ones.

TIBINFO Attributes

The properties defined by TIBINFO and their allowable values are listed below and are described in detail in the appropriate "man" pages. The last grouping of properties allow the programmer to send default property values and hints to the underlying system components—specifically, the network protocol engines, the TIB™ subject mapper, and various service disciplines.

<code>TIB_PROP_CFILE</code>	<code>cfile-handle</code>
<code>TIB_PROP_CLIENTDATA</code>	<code>pointer</code>
<code>TIB_PROP_ERRHANDLER</code>	<code>error-handler-routine</code>
<code>TIB_PROP_LASTMSG</code>	<code>tib_message pointer</code>
<code>TIB_PROP_MSGHANDLER</code>	<code>message-handler-routine</code>
<code>TIB_PROP_NETWORK</code>	<code>protocol-engine-property-list</code>
<code>TIB_PROP_NETWORK_CFILE</code>	<code>protocol-engine-property-cfile</code>
<code>TIB_PROP_SERVICE</code>	<code>service-discipline-property-list</code>
<code>TIB_PROP_SERVICE_CFILE</code>	<code>service-discipline-property-cfile</code>
<code>TIB_PROP_SUBJECT</code>	<code>subject-property-list</code>
<code>TIB_PROP_SUBJECT_CFILE</code>	<code>subject-property-cfile</code>

The component information of a TIBINFO message can be accessed through the following macros:

```
tib_msg_clientdata(msg)
tib_msg_subject(msg)
tib_msg_size(msg)
tib_msg_value(msg)
```

The following macros return TRUE (1) or FALSE (0):

```
tib_msg_is_buffer(msg)
tib_msg_is_form(msg)
```

5. TIB™ Forms

5.1 Introduction

The Forms package provides the tools to create and manipulate self-describing data objects, e.g., forms. Forms have sufficient expressiveness, flexibility and efficiency to describe all data exchanged between the different TIB™ applications, and also between the main software modules of each application.

The Forms package provides its clients with one data abstraction. Hence, the software that uses the Forms package deal with only one data abstraction, as opposed to a data abstraction for each different type of data that is exchanged. Using forms as the only way to exchange user data, facilitates (i) the integration of new software modules that communicate with other software modules, and (ii) modular enhancement of existing data formats without the need to modify the underlying code. This results in software that is easier to understand, extend, and maintain.

Forms are the principal shared objects in the TIB™ communication infrastructure and applications; consequently, one of the most important abstractions in the TIB™.

The primary objective in designing the forms package were:

Extensibility—It is desirable to be able to change the definition of a form class without recompiling the application, and to be able introduce new classes of forms into the system.

Maintainability—Form-class definition changes may affect many workstations; such changes must be propagated systematically.

Expressiveness—Forms must be capable of describing complex objects; therefore, the form package should support many basic types such as integer, real, string, etc. and also sequences of these types.

Efficiency—Forms should be the most common object used for sending information between processes—both for processes on the same workstation and for processes on different workstations. Hence, forms should be designed to allow the communication infrastructure to send information efficiently.

Note that our use of the term "form" differs from the standard use of the term in database systems and so-called "forms management systems." In those systems, a "form" is a format for displaying a database or file record. (Typically, in such systems, a user brings up a form and paints a database record into the form.)

Our notion of a form is more fundamental, akin to such basic notions as record or array. Our notion takes its meaning from the original meaning of the Latin root word *forma*.

Borrowing from Webster: "The shape and structure of something as distinguished from its material". Forms can be instantiated, operated on, passed as arguments, sent on a

network, stored in files and databases. Their contents can also be displayed in many different formats. "templates" can be used to specify how a form is to be displayed. A single form (more precisely, a form class) can have many "templates" since it may need to be displayed in many different ways. Different kinds of users may, for example, desire different formats for displaying a form.

5.2 Description

Forms are self-describing data objects. Each form contains a reference to its formclass, which completely describes the form. Forms also contains metadata that enables the form package to perform most operations without accessing the related formclass definition.

Each form is a member of a specific form class. All forms within a class have the same fields and field's labels (in fact, all defining attributes are identical among the forms of a specific class). Each form class is named and two classes are considered to be distinct if they have distinct names (even though the classes may have identical definitions). Although the forms software does not assign any special meaning or processing support to particular form names, the applications using it might. (In fact, it is expected that certain form naming conventions will be established.)

There are two main classification of forms: primitive versus constructed forms, and fixed length versus variable size forms.

Primitive forms are used to represent primitive data types such as integers, float, strings, etc. Primitive forms contain metadata, in the form header information header and the data of the appropriate type, such as integer, string, etc.

Constructed forms contain sub-forms. A constructed form contains other forms, which in turn can contain subforms.

Fixed length forms are simply forms of a fixed length, e.g., all the forms of a fixed length class occupy the same number of bytes. An example for a fixed length primitive form is the integer form class; integer forms always take 6 bytes, (2 bytes for the form header and 4 bytes for the integer data).

Variable size forms contain variable size data: variable size, primitive forms contain variable size data, such as variable length string; variable size, constructed forms contain a variable number of subforms of a single class. Such forms are similar to an array of elements of the same type.

5.3 Class Identifiers

When a class is defined it is assigned an identifier. This identifier is part of each of the class's form instance, and is used to identify the form's class. This identifier is in addition to its name. Class identifiers must be unique within their context of use. Class identifiers are 2 bytes long; bit 15 is set if the class is fixed length and cleared otherwise; bit 14 is set if the class is primitive and cleared otherwise;

5.4 Assignment Semantics

To assign and retrieve values of a form (or a form sequence), "copy" semantics is used. Assigning a value to a form (form field or a sequence element) copies the value of the form to the assigned location-it does not point to the given value.

Clients that are interested in pointer semantics should use forms of the basic type Form Pointer and the function `Form_set_data_pointer`. Forms of type Form Pointer contain only a pointer to a form; hence, pointer semantics is used for assignment. Note that the C programming language supports pointer semantics for array assignment.

5.5 Referencing a Form Field

A sub-form or field of a constructed form can be accessed by its field name or by its field identifier (the latter is generated by the Forms package). The name of a subform that is not a direct descendant of a given form is the path name of all the fields that contain the requested subform, separated by dot. Note that this is similar to the naming convention of the C language records.

A field identifier can be retrieved given the field's name and using the function `Form_class_get_field_id`. The direct fields of a form can be traversed by using `Form_field_id` first to get the identifier of the first field, and then by subsequently calling `Form_field_id_next` to get the identifiers of each of the next fields.

Accessing a field by its name is convenient; accessing it by its identifier is fast. Most of the Forms package function references a form field by the field's identifier and not by the field's name.

5.6 Form-class Definition Language

Form classes are specified using the "form-class definition language," which is illustrated below. Even complex forms can be described within the simple language features depicted below. However, the big attraction of a formal language is that it provides an extensional framework: by adding new language constructs the descriptive power of the language can be greatly enhanced without rendering previous descriptions incompatible.

A specification of a form class includes the specification of some class attributes, such as the class name, and a list of specifications for each of the class's fields. Three examples are now illustrated:

```

{
short {
    IS_FIXED          true;
    IS_PRIMITIVE      true;
    DATA_SIZE        2;
    DATA_TYPE        9;
}
short_array {          # A variable size class of shorts.
    IS_FIXED          false;
    FIELDS {
        {
            FIELD_CLASS_NAME short;
        }
    }
}
example_class {
    IS_FIXED          true;
    FIELDS {
        first {
            FIELD_CLASS_NAME short;
        }
        second {
            FIELD_CLASS_NAME short_array;
        }
        third {
            FIELD_CLASS_NAME string_30;
        }
        fourth {
            FIELD_CLASS_NAME integer;
        }
    }
}
}

```

To specify a class, the class's name, a statement of fixed or variable size, and a list of fields must be given. For primitive classes the data type and size must also be specified. All the other attributes may be left unspecified, and defaults will be applied. To define a class field, either the field class name or id must be specify.

The form-class attributes that can be specified are:

The class name.

CLASS_ID—The unique short integer identifier of the class. Defaults to a package specified value.

IS_FIXED—Specifies whether its a fixed or variable size class. Expects a boolean value. This is a required attribute.

IS_PRIMITIVE—Specifies whether its a primitive or constructed class. Expects a boolean value. Defaults to False.

FIELDS_NUM—An integer specifying the initial number of fields in the form. Defaults to the number of specified fields.

DATA_TYPE—An integer, specified by the clients, that indicates what is the type of the data. Used mainly in defining primitive classes. It does not have default value.

DATA_SIZE—The size of the forms data portion. Used mainly in defining primitive classes. It does not have default value.

FIELDS—Indicates the beginning of the class's fields definitions.

The field attributes that can be specified are:

The class field name.

FIELD CLASS ID—The class id for the forms to reside in the field. Note that the class name can be used for the same purpose.

FIELD_CLASS_NAME—The class name for the forms to reside in the field.

Here is an example of the definition of three classes:

Note that variable length forms contains fields of a single class. "integer" and "string_30", used in the above examples, are two primitive classes that are defined within the Formclass package itself.

5.7 Form Classes are Forms

Form classes are implemented as forms. This means functions that accept forms as an argument also accept form classes. Some of the more useful functions on form classes are:

Form_pack, Form_unpack—Can be used to pack and unpack form classes.

Form_copy—Can be used to copy form classes.

Form_show—Can be used to print form classes.

5.8 Types

```
typedef Formclass
```

A form-class handle.

```
typedef Formclass_id
```

A form-class identifier.

```
typedef Formclass_attr
```

A form-class attribute type. Supported attributes are:

FORMCLASS_SIZE—The size of the class form instances.

FORMCLASS_NAME—The class name.

FORMCLASS_ID—A two byte long unique identifier.

FORMCLASS_FIELDS_NUM—The number of (direct) fields in the class. This is applicable only for fixed length classes. The number of fields in a variable length class is different for each instance; hence, is kept in each form instance.

FORMCLASS_INSTANCES_NUM—The number of form instances of the given class.

FORMCLASS_IS_FIXED—True if its a fixed length form, False if its a variable length form.

FORMCLASS_IS_PRIMITIVE—True if its a form of primitive type, False if its a constructed form, i.e. the form has sub forms.

FORMCLASS_DATA_TYPE—This field value is assigned by the user of the forms a to identify the data type of primitive forms. In our current application we use the types constants as defined by the enumerated type `Form_data_type`, in the file `forms.h`.

FORMCLASS DATA SIZE—This field contains the data size, in bytes, of primitive forms. For instance, the data size of the primitive class `Short` is two. Because it contains the C type `short`, which is kept in two bytes.

```
typedef Formclass_field_attr
```

A form-class field attribute type. Supported form class field attributes are:

FORMCLASS_FIELD_NAME—The name of the class field.

FORMCLASS_FIELD_CLASS_ID—The class id of the field's form.

```
typedef Form
```

A form handle.

```
typedef Form_field_id
```

An identifier for a form's field. It can identifies fields in any level of a form. A field identifier can be retrieved form a field name, using the function `Form-class_get_field_name`. A `form_field_id` is manipulated by the functions:

`Form_field_id_first` and `Form_field_id_next`.

```
typedef Form_attr
```

A form attribute type. Supported form attributes are:

FORM_CLASS_ID—The form's class identifier.

FORM_DATA_SIZE—The size of the form's data. Available only for constructed, not primitive, forms or for primitive forms that are of variable size. For fixed length primitive forms this attribute is available via the form class.

FORM_FIELDS_NUM—The number of fields in the given form. Available only for constructed, not primitive forms. For primitive forms this attribute is available via the form class.

```
typedef Form_data
```

The type of the data that is kept in primitive forms.

```
typedef Form_pack_format
```

Describes the possible form packing types. Supported packing formats are:

FORM_PACK_LIGHT—Light packing, used mainly for inter process communication between processes on the same machine. It is more efficient then other types of packing. Light packing consists of serializing the given form, but it does not translates the form data into machine independent format.

FORM_PACK_XDR—Serialize the form while translating the data into a machine-independent format. The machine-independent format used is Sun's XDR.

5.9 Procedural Interface to the Forms-class Package

The formclass package is responsible for creating and manipulating forms classes. The forms package uses these

69

descriptions to create and manipulate instances of given form classes. An instance of a form class is called, not surprisingly, a form.

Formclass_create

Create a class handle according to the given argument list. If the attribute `CLASS_CFILE` is specified it should be followed by a cfile handle and a path_name. In that case `formclass_create` locates the specification for the form class in the specified configuration file. The specification is compiled into an internal data structure for use by the forms package.

`Formclass_create` returns a pointer to the class data structure. If there are syntax errors in the class description file the function sets the error message flag and returns `NULL`.

Formclass_destroy

The class description specified by the given class handle is dismantled and the storage is reclaimed.

If there are live instances of the class then the class is not destroyed and the error value is updated to "FORM-CLASS_ERR_NON_ZERO_INSTANCES_NUM".

Formclass_get

Given a handle to a form class and an attribute of a class (e.g. one of the attributes of the type `Formclass_attr`) `Formclass_get` returns the value of the attribute. Given an unknown attribute the error value is updated to "FORM-CLASS_ERR_UNKNOWN_ATTRIBUTE".

Formclass_get_handle_by_id

Given a forms-class id, `Formclass_get_handle_by_id` returns the handle to the appropriate class descriptor. If the requested class id is not known `Form-class_get_handle_by_id` returns `NULL`, but does not set the error flag.

Formclass_get_handle_by_name

Given a forms-class name, `Formclass_get_handle_by_name` returns the handle to the appropriate class descriptor. If the requested class name is not known `Formclass_get_handle_by_name` returns `NULL`, but does not set the error flag.

Formclass_get_field_id

Given a handle to a form class and a field name this function returns the form id, which is used for a fast access to the form. If the given field name does not exist, it updated the error variable to `FORMCLASS_ERR_UNKNOWN_FIELD_NAME`.

Formclass_field_get

Returns the value of the requested field's attribute. If an illegal id is given this procedure, it updated the error variable to `FORMCLASS_ERR_UNKNOWN_FIELD_ID`.

Formclass_iserr

Returns `TRUE` if `Formclass` error flag is turned on, `FALSE` otherwise.

70

Formclass_errno

Returns the formclass error number. If no error, it returns `FORMCLASS_OK`. For a list of supported error values see the file `formclass.h`.

5.10 The Forms Package

Form_create

Generate a form (i.e., an instance) of the form class specified by the parameter and return a handle to the created form.

Form_destroy

The specified form is "destroyed" by reclaiming its storage.

Form_get

Given a handle to a form and a valid attribute (e.g. one of the values of the enumerated type `Form_attr`) `Form_get` returns the value of the requested attribute.

The attribute `FORM_DATA_SIZE` is supported only for variable size forms. For fixed size form this information is kept in the class description and is not kept with each form instance.

Requiring the `FORM_DATA_SIZE` from a fixed length form will set the error flag to `FORM_ERR_NO_SIZE_ATTR_FOR_FIXED_LENGTH_FORM`.

The attribute `FORM_FIELDS_NUM` is supported only for constructed forms. Requiring the `FORM_FIELDS_NUM` from a primitive form will set the error flag to `FORM_ERR_ILLEGAL_ATTR_FOR_PRIMITIVE_FORM`.

If the given attribute is not known the error flag is set to `FORM_ERR_UNKNOWN_ATTR`. When the error flag is set differently, then `FORM_OK` `Form_get` returns `NULL`.

Form_set_data

Sets the form's data value to the given value. The given data argument is assumed to be a pointer to the data, e.g., a pointer to an integer or a pointer to a date structure. However for strings we expect a pointer to a character.

Note that we use Copy semantics for assignments.

Form_get_data

Return a pointer to form's data portion. In case of a form of a primitive class the data is the an actual value of the form's type. If the form is not of a primitive class, i.e., it has a non zero number of fields, then the form's value is a handle to the form's sequence of fields.

Warning, the returned handle points to the form's data structure and should not be altered. If the returned value is to be modified is should be copied to a private memory.

Form_set_data_pointer

Given a variable size form, `Form_set_data_pointer` assigns the given pointer to the points to the forms data portion. `Form_set_data_pointer` provide a copy operation with pointer semantics, as opposed to copy semantics.

If the given form is a fixed length form then the error flag is set to `FORM_ERR_CANT_ASSIGN_POINTER_TO_FIXED_FORM`.

71

Form_field_set_data

This is a convenient routine that is equal to calling Form_field_get and then using the retrieved form to call Form_set_data. More precisely: form_field_set_data(form, field_id, form_data, size) = form_set_data(form_field_get(form, field_id), form_data, size), plus some error checking.

Form_field_get_data

Note that we use Copy semantics for assignments.

This is a convenient routine that is equal to calling Form_field_get and then using the retrieved form to call Form_get_data. More precisely: form_field_get_data(form, field_id, form_data, size) = form_get_data(form_field_get(form, field_id), form_data, size) plus some error checking.

Warning, the returned handle points to the form's data structure and should not be altered. If the returned value is to be modified it should be copied to a private memory.

form_field_id_first

Form_field_id_first sets the given field_id to identify the first direct field of the given form handle.

Note that the memory for the given field_id should be allocated (and freed) by the clients of the forms package and not by the forms package.

form_field_id_next

Form_field_id_first sets the given field_id to identify the next direct field of the given form handle. Calls to Form_field_id_next must be preceded with a call to Form_field_id_first.

Note that the memory for the given field_id should be allocated (and freed) by the clients of the forms package and not by the forms package.

Form_field_set

Sets the given form or form sequence as the given form field value. Note that we use Copy semantics for assignments.

When a nonexistent field id is given then the error flag is set to FORM_ERR_ILLEGAL_ID.

Form_field_get

Return's a handle to the value of the requested field. The returned value is either a handle to a form or to a form sequence.

Warning, the returned handle points to the form's data structure and should not be altered. If the returned value is to be modified it should be copied to a private memory, using the Form_copy function.

When a nonexistent field id is given, then the error flag is set to FORM_ERR_ILLEGAL_ID and Form_field_get returns NULL.

Form_field_append

Form_field_append appends the given, append_form argument to the end of the base_form form sequence. Form_field_append returns the id of the appended new field.

72

Form_field_delete

Form_field_delete deletes the given field from the given base_form.

If a non existing field id is given then the error flag is set to FORM_ERR_ILLEGAL_ID and Form_field_delete returns NULL.

Form_pack

Form_pack returns a pointer to a byte stream that contains the packed form, packed according to the requested format and type.

If the required packed type is FORM_PACK_LIGHT then Form_pack serializes the form, but the form's data is not translated to a machine-independent representation. Hence a lightly packaged form is suitable to transmit between processes on the same machine.

If the required packed type is FORM_PACK_XDR then Form_pack serializes the form and also translates the form representation to a machine-independent representation, which is Sun's XDR. Hence form packed by an XDR format are suitable for transmitting on a network across machine boundaries.

Formclass.h. are implemented as forms, hence Form_pack can be used to pack form classes as well as forms.

Form_unpack

Given an external representation of the form, create a form instance according to the given class and unpack the external representation into the instance.

Form classes are implemented as forms, hence Form_unpack can be used to unpack form classes as well as forms.

Form_copy

Copy the values of the source form into the destination form. If the forms are of different classes no copying is performed and the error value is updated to FORM_ERR_ILLEGAL_CLASS.

Formclasses are implemented as forms, hence Form_copy can be used to copy form classes as well as forms.

Form_show

Return an ASCII string containing the list of field names and associated values for indicated fields. The string is suitable for displaying on a terminal or printing (e.g., it will contain new-line characters). The returned string is allocated by the function and need to be freed by the user. (This is function is very useful in debugging.)

Formclasses are implemented as forms, hence Form_show can be used to print form classes as well as forms.

Form_iserr

Returns TRUE if the error flag is set, FALSE otherwise.

Form_errno

Returns the formclass error number. If no error, it returns FORMCLASS_OK. The possible error values are defined in the file forms.h.

GLOSSARY

There follows a list of definitions of some of the words and phrases used to describe the invention.

Access Procedure: a broader term than service discipline or service protocol because it encompasses more than a communications protocol to access data from a particular server, service, application. It includes any procedure by which the information requested on a particular subject may be accessed. For example, if the subject request is "Please give me the time of date", the access procedure to which this request is mapped on the service layer could be a call to the operating system on the computer of the user that initiated the request. An access procedure could also involve a call to a utility program.

Application: A software program that runs on a computer other than the operating system programs.

Architectural Decoupling: A property of a system using the teachings of the invention. This property is inherently provided by the function of the information layer in performing subject-based addressing services in mapping subjects to services and service disciplines through which information on these subjects may be obtained. Subject-based addressing eliminates the need for the data consuming processes to know the network architecture and where on the network data on a particular subject may be found.

Attribute of a Form Class: A property of form class such as whether the class is primitive or constructed. Size is another attribute.

Class: A definition of a group of forms wherein all forms in the class have the same format and the same semantics.

Class/Class Descriptor/Class Definition: A definition of the structure and organization of a particular group of data records or "forms" all of which have the same internal representation, the same organization and the same semantic information. A class descriptor is a data record or "object" in memory that stores the data which defines all these parameters of the class definition. The Class is the name of the group of forms and the Class Definition is the information about the group's common characteristics. Classes can be either primitive or constructed. A primitive class contains a class name that uniquely identifies the class (this name has associated with it a class number or class_id) and a specification of the representation of a single data value. The specification of the representation uses well known primitives that the host computer and client applications understand such as string_20 ASCII, floating point, integer, string_20 EBCDIC etc. A constructed class definition includes a unique name and defines by name and content multiple fields that are found in this kind of form. The class definition specifies the organization and semantics of the form by specifying field names. The field names give meaning to the fields. Each field is specified by giving a field name and the form class of its data since each field is itself a form. A field can be a list of forms of the same class instead of a single form. A constructed class definition contains no actual data although a class descriptor does in the form of data that defines the organization and semantics of this kind of form. All actual data that define instances of forms is stored in forms of primitive classes and the type of data stored in primitive classes is specified in the class definition of the primitive class. For example, the primitive class named "Age" has one field of type integer_3 which is defined in the class definition for the age class of forms. Instances of forms of this class contain 3 digit integer values.

Class Data Structure: All the data stored in a class manager regarding a particular class. The class descriptor is the most important part of this data structure, but there may be more information also.

Class Definition: The specification of a form class.

Class Descriptor: A memory object which stores the form-class definition. In the class manager, it is stored as a form. On disk, it is stored as an ASCII string. Basically, it is a particular representation or format for a class definition. It can be an ASCII file or a form type of representation. When the class manager does not have a class descriptor it needs, it asks the foreign application that created the class definition for the class descriptor. It then receives a class descriptor in the format of a form as generated by the foreign application. Alternatively, the class manager searches a file or files identified to it by the application requesting the semantic-dependent operation or identified in records maintained by the class manager. The class definitions stored in these files are in ASCII text format. The class manager then converts the ASCII text so found to a class descriptor in the format of a native form by parsing the ASCII text into the various field names and specifications for the contents of each field.

Client Application: a data consuming or data publishing process, i.e., a computer program which is running, other than an operating system program that is linked to the communication interface according to the teachings of the invention.

Computer NetWork: A data pathway between multiple computers by hardware connection such as a local or wide area network or between multiple processes running on the same computer through facilities provided by the operating system or other software programs and/or shared memory including a Unix pipe between processes.

Configuration Decoupling: The property of a computer system/network implementing the teachings of the invention which is inherently provided by the distributed communication layer. This layer, by encapsulating the detailed protocols of how to set up and destroy communication links on a particular configuration for a computer network, frees client processes, whether data publishers or data consumers from the need to know these details.

Configuration File: A file that stores data that describes the properties and attributes or parameters of the various software components, records and forms in use.

Constructed Field: A field which contains another form or data record.

Consumer: a client or consumer application or end user which is requesting data.

Data Distribution Decoupling: The function of the communication interface software according to the teachings of the invention which frees client applications of the necessity to know and provide the network addresses for servers providing desired services.

Decoupling: Freeing a process, software module or application from the need to know the communication protocols, data formats and locations of all other processes, computers and networks with which data is to be interchanged.

Distributed Communication Layer: the portion of the apparatus and method according to the teachings of the invention which maps the access procedure identified by the service layer to a particular network or transparent layer protocol engine and sets up the required communication channel to the identified service using the selected network protocol engine.

Field: One component in an instance of a form which may have one or more components each named differently and each meaning a different thing. Fields are "primitive" if they contain actual data and are "constructed" if they contain other forms, i.e., groupings of other fields. A data record or form which has at least one field which contains

another form is said to be "nested". The second form recorded in the constructed field of a first form has its own fields which may also be primitive or constructed. Thus, infinitely complex layers of nesting may occur.

Foreign: A computer or software process which uses a different format of data record than the format data record of another computer or software process.

Form: A data record or data object which is self-describing in its structure by virtue of inclusion of fields containing class descriptor numbers which correspond to class descriptors, or class definitions. These class descriptors describe a class of form the instances of which all have the same internal representation, the same organization and the same semantic information. This means that all instances, i.e., occurrences, of forms of this class have the same number of fields of the same name and the data in corresponding fields have the same representation and each corresponding field means the same thing. Forms can be either primitive or constructed. A form is primitive if it stores only a single unit of data. A form is constructed if it has multiple internal components called fields. Each field is itself a form which may be either primitive or constructed. Each field may store data or the class_id, i.e., the class number, of another form.

Format Operation: An operation to convert a form from one format to another format.

Format or Type: The data representation and data organization of a structural data record, i.e., form.

Handle: A pointer to an object, record, file, class descriptor, form etc. This pointer essentially defines an access path to the object. Absolute, relative and offset addresses are examples of handles.

ID: A unique identifier for a form, record, class, memory object etc. The class numbers assigned the classes in this patent specification are examples of ID's.

Information Layer: the portions of the apparatus and method according to the teachings of the invention which performs subject based addressing by mapping information requests on particular subjects to the names of services that supply information on the requested subject and the service disciplines used to communicate with these services.

Interface: A library of software programs or modules which can be invoked by an application or another module of the interface which provide support functions for carrying out some task. In the case of the invention at hand, the communication interface provides a library of programs which implement the desired decoupling between foreign processes and computers to allow simplified programming of applications for exchanging data with foreign processes and computers.

Interface Card: The electronic circuit that makes a physical connection to the network at a node and is driven by transparent layer protocol programs in the operating system and network and data-link protocol programs on the interface card to send and receive data on the network.

Native Format/Form: The format of a form or the form structure native to an application and its host computer.

Nested: A data structure comprised of data records having multiple fields each of which may contain other data records themselves containing multiple fields.

Network Protocol Engine: a software and hardware combination that provides a facility whereby communication may be performed over a network using a particular protocol.

Node: Any computer, server or terminal coupled to the computer network.

Primitive Field: A field of a form or data record which stores actual data.

Process: An instance of a software program or module in execution on a computer.

Semantic-Dependent Operation: An operation requiring access to at least the semantic information of the class definition for a particular form to supply data from that form to some requesting process.

Semantic Information: With respect to forms, the names and meanings of the various fields in a form.

Server: A computer running a data producer process to do something such as supply files stored in bulk storage or raw data from an information source such as Telerate to a requesting process even if the process is running on the same computer which is running the data producer process.

Server Process: An application process that supplies the functions of data specified by a particular service, such as Telerate, Dow Jones News Service, etc.

Service: A meaningful set of functions or data usually in the form of a process running on a server which can be exported for use by client applications. In other words, a service is a general class of applications which do a particular thing, e.g., applications supplying Dow Jones News information. Quotron datafeed or a trade ticket router. An application will typically export only one service, although it can export many different services.

Service Discipline or Service Protocol: A program or software module implementing a communication protocol for communication with a particular service and including routines by which to select one of several servers that supplies a service in addition to protocols for communicating with the service and advising the communication layer which server was selected and requesting that a communication link be set up.

Service Access Protocol: A subset of the associated service discipline that encapsulates a communication protocol for communicating with a service.

Service Instance: A process running on a particular computer and which is capable of providing the specified service (also sometimes called a server process). For a given service, several service instances may be concurrently providing the service so as to improve performance or to provide fault tolerance. The distributed communication component of the TIB™ communication software implements "fault-tolerant" communication by providing automatic switchover from a failed service instance to an operational one providing the same service.

Service Layer: the portion of apparatus and method according to the teachings of the invention that maps data received from the information layer to the access procedure to be used to access the service or other source for the requested information to provide service decoupling.

Service Decoupling: The function of the service layer of the communication interface software according to the teachings of the invention which frees client applications of the necessity to know and be able to implement the particular communication protocols necessary to access data from or otherwise communicate with services which supply data on a particular subject.

Service Record: A record containing fields describing the important characteristics of an application providing the specified service.

Subject Domain: A set of subject categories (see also subject space).

Subject Space: A hierarchical set of subject categories.

Subscribe Request: A request for data regarding a particular subject which does not specify the source server or servers, process or processes or the location of same from which the data regarding this subject may be obtained.

Transport Layer: A layer of the standard ISO model for networks between computers to which the communication interface of the invention is linked.

Transport Protocol: The particular communication protocol or discipline implemented on a particular network or group of networks coupled by gateways or other inter-network routing.

Appendix A

Appendix A, which can be found in the application file, is the complete run time collection of C language source code programs that embody the invention. This code runs in the Sun Microsystems OS 4.1 environment on Sun 4 machines such as SPARC 1 and the SPARC 370 and 470 servers.

TIB Build

TIB may be built on Sun Unix workstations running Sun OS 4.1. The C compiler distributed by Sun is used for compilation. Also required are the X11R3 (or X11R4) libraries. TIB was built using the "gmake" utility (distributed by the Free Software Foundation in Cambridge, Mass.) Each directory which must be built contains gmake file named GNUmakefile. To build the API, three invocations of gmake should be performed on each directory containing a GNUmakefile. The first pass directs gmake to build "include files" by running 'gmake EXPORT-INCS'. The second pass directs gmake to build the TIB libraries by running 'gmake EXPORT-LIBS'. The last pass through the directories directs gmake to build the TIB API libraries by running 'gmake EXPORT-BINS'.

Several gmake support files exist, and a directory should be created to hold these files. The environment variable, \$MINCLUDE, should be set to reference this directory before performing the build.

What is claimed is:

1. In a distributed computing environment including one or more computers coupled together by one or more networks, said one or more computers having in execution thereon one or more subscriber applications each of which has the capability of making a subscription request requesting that data on one or more subjects be sent to said subscriber application whenever data on said subject becomes available until said subscription is canceled, and said one or more computers having in execution thereon one or more service applications, each of which is capable of transmitting over said one or more networks data on one or more subjects, an apparatus comprising:

intermediary software controlling said one or more computers and coupled to said one or more networks and said one or more subscriber and service applications, so as to logically decouple said one or more subscriber applications from said service applications in the sense of performing all necessary functions to get data requested by subject by said subscriber applications from one or more service applications that supply data on the requested subject to all computers on which a subscriber application is in execution which has an open subscription to the subject of said data and from each such computer to the subscribing applications themselves and delivering said data only to said computers on which there is in execution a subscriber application having an open subscription to the subject of said data, said logical decoupling also implemented by said intermediary software by eliminating the need to have computer code in any said service application

which outputs any data, other than the subject itself, which controls where in said distributed computing environment data published by said service application in execution on one or more of said computers is distributed or which identifies or locates any subscriber application or computer in said distributed computing environment to which data published by any said service application is to be distributed, and by eliminating the need for any subscriber application to include computer code which identifies any particular service application from which data is to be sought or which can find a service application which can supply data on a subject desired by said subscriber application, said intermediary software for controlling said one or more computers so as to automatically set up a communication path through said network between service applications which publish data on a subject and all and only computers having in execution thereon subscriber applications having an open subscription to the subject of said data.

2. An apparatus for use in a distributed computing system comprised of one or more computers some of which may be clients and some of which may be servers and some of which may be both clients and servers, all said computers coupled by a network or data path, said apparatus for interfacing at least one application process in execution on at least one client computer having an operating system embodying a transport layer communication protocol and a network interface circuit card coupling the client computer to said network to which is coupled at least one server computer having an operating system embodying a transport layer protocol and execution of which is controlled by at least one service process, such that said at least one application process requesting data on a subject can be selectively linked to at least one service process which is capable of supplying data on said subject using a subscription paradigm, and wherein each said at least one application process is capable of issuing subscription requests requesting an ongoing flow of data on the subject named in each said subscription request whenever said data is published, comprising:

communication means including at least one protocol engine for execution on each of said host and server computers and coupled to said transport layer communication protocols of said host and server computers, each protocol engine encapsulating a communication protocol providing reliability or efficiency enhancing functions not supplied by said transport layer communication protocol, said protocol engines for exchanging data with each other over said network, said communication means for receiving a link request to set up a subscription-based data communication link to a selected server computer and for selecting and invoking selected ones of said protocol engines encapsulating a desired communication protocol and for invoking said transport layer communication protocol to physically establish said data communication link;

service means for processing subscription requests, said service means including at least two service discipline(s) including a consumer side service discipline program for controlling one or more of said computers so as to communicate with said at least one application process and a publisher side service discipline program for controlling one or more of said computers so as to communicate with said at least one service process and which is capable of receiving data published on a subject by said service process and distributing said

published data to only those computers having in execution thereon an application process having an open subscription to the subject of said data, said data published by said service process containing no data indicating where said data is to be sent within said distributed computing system, other than the subject of the published data itself, said service means for transmitting said data to all application processes which have issued subscription requests naming the subject of said data as being of interest, and wherein at least said consumer side service discipline encapsulates a communication protocol for communicating to at least a selected one of said services via the publisher side service discipline coupled thereto, said service means for receiving a subscription link request requesting establishment of a subscription communication link with one or more of said service processes named in said subscription communication link which publishes data on a particular subject and for generating and transmitting to said communication means said link request requesting the establishment by said communication means of said subscription-based data communication link with the service identified in said subscription link request, and for sending a subscription registration message to said publisher side service discipline coupled to said one or more service processes requesting that all data on said subject, whenever published, be sent to each said application process(es) which has issued a subscription request on the subject so long as said subscription has not been canceled, said publisher side service discipline being selectively coupled to said service process for registering said subscription in a subscription list identifying at least all application processes that have active subscriptions and the subjects thereof, at least one of said service disciplines being coupled to each said application and services process for which an active subscription and communication link have been established and cooperating with each said application process that issued a subscription request and at least one said service process that is capable of supplying data on the subject so as to forward data published by said service process(es) on the subject of an active subscription to all said application processes having an active subscription on said subject, said service means for filtering said published data by subject on the service side of the data communication link such that data published by said service process which does not correspond to the subject of any active subscription is not transmitted over the network but such that data having a subject matching the subject of an active subscription reaches all application processes which have active subscriptions on the subject; and

information means for controlling one or more computers so as to present a programmatic interface at least to said one or more application processes and said at least one service processes such said one or more application processes can open a subscription to a subject simply by passing a subscribe request to said programmatic interface and passing a subject thereto in said subscription request and such that a service process can publish data on a subject to all said application processes having an open subscription to said subject simply by invoking a publish function of said programmatic interface and passing the data to be published thereto along with a subject of said data, said information means for receiving said subscribe request(s) from said one or

more application process(es), each said subscribe request containing at least a subject and the identity of a callback routine for said application process, and, for each said subscribe request, mapping said subject contained therein to one or more of said service processes which can supply data on said subject and at least a selected one of said service disciplines which encapsulates a communication protocol capable of communicating with said selected service or services, and for transmitting the subject of said subscribe request to said service means along with data identifying and/or giving the address in said distributed computing system of at least one service process with which subscription based data communication links are to be established.

3. The apparatus of claim 2 wherein data transmitted between said application process and said service process is transmitted over said data path or network as a data message comprising a plurality of sequential packets, each of which has a header and error corrections bits as part of said packet, and wherein said communication means further comprises:

a memory;

means for adding reliability enhancing sequence numbers to said packet headers prior to transmission;

means for transmitting over said data path or network the packets, error correction bits and sequence numbers which comprise a complete data message;

means for saving said packets in said memory with said sequence numbers and error correction bits in case some or all of the packets need to be retransmitted;

means for transmitting said packets;

means for receiving said packets and checking said reliability sequence numbers to determine that all packets have arrived and using said error correction bits to determine if any of said packets were garbled during transmission, and for requesting retransmission of any missing or garbled packets, or acknowledging that all packets were properly received; and

means for retransmitting any missing or garbled packets and for flushing any packets no longer needed from said memory.

4. In a distributed computing environment including at least one data consuming process in execution on one or more computers and at least one data producing process in execution on one or more computers, said one or more computers and said data consuming and data producing processes coupled by at least one data path and/or network, an apparatus comprising:

one or more computers in said distributed computing environment processing of which is controlled by one or more subject based addressing programs, said one or more subject based addressing programs for controlling said one or more computers to receive a subscription request from one or more data consuming processes to locate and access data on a specified subject, and for controlling said one or more computers to automatically locate at least one data producing process which produces data on said subject and for controlling said one or more computers to automatically generate each of said one or more link requests requesting that one or more subscription communication links be established over said data path between at least one of said data producing processes which supply data on the subject of said subscription request and at least one of said data consuming processes which made subscription requests for data on said subject; and

one or more computers in said distributed computing environment controlled by one or more communication

programs to receive said link requests from said one or more computers controlled by said subject based addressing programs, and for controlling said one or more computers so as to automatically establish said one or more subscription communication links over said data path, each of said one or more subscription communication links being established using a communication protocol appropriate for communication with the data producing process with which said subscription communication link is established, said one or more subscription communication links being established by controlling said one or more computers controlled by said one or more communication programs such that there is no need for said one or more computers controlled by said one or more communication programs to receive any address or address related data, other than the subject itself of the subscription request, from any data consuming process indicating where in said distributed computing environment a data producing process which is publishing data on the subject may be found and by controlling said one or more computers controlled by said one or more communication programs such that there is no need to receive any data from any data producing process, other than the subject itself of the data published by said data producing process, indicating or controlling where the data published by the data producing process is to be sent in said distributed computing environment, said one or more communication programs also for controlling said one or more computers so as to automatically transport data published on the subjects of any active subscription(s) to all but only said one or more computers processing of which is controlled by a data consuming processes which issued a subscription request requesting data on said subject and for controlling said one or more computers so as to automatically route said data published on a subject for which there is an active subscription to all said one or more data consuming processes having an active subscription on the subject of said data until said subscription(s) is canceled.

5. The apparatus of claim 4 wherein said subject based addressing program includes directory services means for storing data identifying which data producing processes supply data on which subjects and for searching said data in response to receipt of a subscription request and returning any data located during said search identifying or giving the address in said distributed computing environment of one or more data producing processes which can supply data on the subject of said subscription request to said one or more computers controlled by said one or more subject based addressing programs for use in generating said one or more link requests.

6. The apparatus of claim 4 wherein each said subject can have multiple parts which are arranged into a subject space containing 30 or more multiple part subjects, and wherein said subject space is arranged hierarchically based upon said multiple parts of said subjects, and wherein said one or more computers controlled by said one or more subject based addressing programs include means for locating all data producing processes which publish data having a subject which satisfies all parts of said multiple part subject.

7. The apparatus of claim 4 wherein said subject contained within said subscription request can have multiple pads, and wherein each said multiple part subject can include empty strings or other wild card notations, and wherein said one or more computers controlled by said one

or more subject based addressing programs includes means for locating all data producing processes which supply data having a subject which satisfies all parts of said multiple part subject which are not empty strings or wild card notations, and wherein no data producing process controlling any of said one or more computers so as to publish data on one or more subjects includes any software routine, program or data the content of which is dependent upon the existence or location in said distributed computing environment of any data consuming process which can issue subscription request or the purpose of which is to directly or indirectly locate any data consuming process which can issue subscription request, and wherein no data producing process controlling any of said one or more computers so as to publish data on one or more subjects includes any software routine, program or subroutine which functions to assist in any way, other than outputting the subject of published data, in routing data between any said data producing process which controls said one or more computers so as to publish data and any data consuming process controlling one or more computers, and wherein no data consuming process controlling one or more computers includes any software routine, program or the content of which is dependent upon the existence or location of any data producing process or the purpose of which is to control one or more of said computers so as to locate any data producing process other than by outputting the subject itself of the subscription request.

8. The apparatus of claim 6 wherein said multiple parts of said subject are hierarchically arranged, and wherein there are at least 30 data producing and data consuming processes controlling processing by one or more of said computers in said distributed computing environment.

9. The apparatus of claim 7 wherein said multiple pads of said subject are hierarchically arranged.

10. The apparatus of claim 5 wherein said one or more communication programs include means coupled to each of said one or more computers under control of one or more of said data producing processes for keeping a subscription list of all of said one or more data consuming processes which have requested subscriptions and the subjects of said subscriptions and for filtering data published by said one or more computers under control of said one or more data publishing processes such that only data having a subject which matches the subject of an active subscription is transmitted over said network or data path.

11. The apparatus of claim 6 wherein each said computer under control of one of said communication programs further comprise means coupled to each data consuming process and each data producing process via the one or more computers controlled thereby for keeping a list indicating for each subject whether there are or are not active subscriptions for that subject, and wherein each said computer under control of one of said communication programs further comprises means for checking said list each time a data message on any subject is published by one or more computers under control of one or more of said data producing processes and for broadcasting over said data path or network each data message published by one or more computers under control of any data producing process on a subject for which there is at least one active subscription indicated on said list, and each said computer under control of at least one of said communication programs further comprising a communication daemon means for controlling execution by each said computer by multitasking so as to cause each said computer under control of at least one of said communications means to filter incoming data messages by subject so

that data messages on any subject for which there is an active subscription entered by a data consuming process controlling processing in a multitasking environment on one of said computers on which a communication daemon means is also controlling execution of said computer on a multitasking basis are automatically sent to all data consuming processes in execution on said computer that requested said data and all other data messages on subjects for which there are no active subscriptions by any data consuming process controlling execution of said computer on a multitasking basis are discarded.

12. The apparatus of claim 8 wherein said one or more computers under control of said one or more communication programs further comprises means coupled to each data consuming process for filtering incoming data by subject so that only data on the requested subject reaches the data consuming process which requested said data.

13. The apparatus of claim 4 further comprising a distributed communications component comprised of one or more computers controlled by one or more computer programs to receive instructions from said one or more computers controlled by one or more subject based addressing programs to establish communications with one or more computers controlled by one or more data publishing computer programs, said distributed communications component for shielding said application and data publishing computer programs from the need to have routines or computer instructions therein to control said one or more computers so as to be able to know the physical distribution of other computers and computer programs in said distributed system and the communication processes so as to be able to communicate with said other computers and computer programs thereby insulating said application and data publishing computer programs from having dependencies on various characteristics of said distributed system designed into the computer programming instructions of said application and data publishing computer programs which could render said application and data publishing computer programs obsolete when said characteristics of said distributed system change.

14. In a computing environment having one or more computers coupled by one or more data paths and/or one or more networks, and having one or more data producing processes which publish data messages on one or more subjects arranged into a subject space, and one or more data consuming processes which need data on one or more subjects, said one or more data producing processes and said one or more data consuming processes in execution on said one or more computers, a process comprising the steps of:

storing data encoding the subjects in said subject space in a memory or other means for storing data coupled to of one or more of said computers, and storing in said memory or said other means for storing data mapping data which maps the identity and/or the address in said computing environment of one or more of said data producing processes for execution on one or more of said computers in said distributed computing system which can supply data on a subject for one or more subjects in said subject space;

receiving a subscription request from one or more data consuming processes for data on a particular subject and automatically locating all data producing processes which can supply data messages on that subject by using the subject of said subscription request to search said mapping data to locate a data producing process which can supply data on that subject without any assistance from said one or more data consuming

processes other than receipt of the subject of the subscription request, and generating a link request, said link request directly or indirectly identifying said at least one data producing process located by search of said mapping data which can supply data on the requested subject included in said subscription request, said link request requesting that a communication link be established between at least one of said data producing processes identified in said link request and all of said data consuming processes which requested data on said subject; and

receiving said link request and automatically setting up a communications link with said at least one data producing process identified directly or indirectly in said link request, and directing data messages received from any said data producing process on said subject to all and only computers having their processing controlled by said data consuming processes which requested data on said subject without receiving any data from any said data producing process which publishes a data message, other than the subject of the data itself, which indicates or controls to which computers in said computing environment any said data message is to be sent, and wherein no data producing process needs to include any data or computer instructions the purpose of which is to identify or locate any said data consuming process other than to output the subject of the data message being published.

15. The process of claim 14 wherein said step of automatically setting up said communications link further comprises the step of sending a request for a subscription to said at least one data producing process identified in said link request, said subscription request identifying directly or indirectly the data consuming process or processes which requested data on said subject, and wherein said step of directing data messages published by said data producing process on each said subject includes the step of directing all data messages published by said at least one data producing process to all data consuming processes which have an active subscription for data on said subject, said directing of data messages on a subject to all data consuming processes having active subscriptions for data on said subject occurring continuously each time a data message on said subject is published but not transmitting any message on any subject to any computer in said communication environment not having in execution thereon a data consuming process having an active subscription to the subject of said message, but ceasing to direct data messages to any data consuming process which has canceled its subscription.

16. The process of claim 14 wherein the step of locating all data producing processes which can supply data messages on the subject of a subscription request includes the steps of passing the subject of said subscription request to said one or more computers controlled by said directory services program which causes said one or more computers controlled by said directory services program to search said mapping data in said data stored in said computer memory encoding said subject space to locate the subject of said subscription request and return data identifying and/or giving the address in said computing environment of at least one data producing process which can supply data on the subject if any such process exists, and wherein said data returned by said computer controlled by said directory services programs is used to generate said link request, and wherein the step of automatically setting up a communications link further comprises the step of making one or more subscription lists listing subjects for which there are active

subscriptions, said one or more subscription lists directly or indirectly identifying the data consuming processes that initiated the active subscriptions, and wherein the step of directing data messages on a subject to all data consuming processes which requested data on said subject further comprises the steps of comparing the subject of each data message published by any data publishing process to the subjects having active subscriptions listed on said one or more subscription lists and directing any data message published on a subject for which there is an active subscription to all data consuming processes that requested data on the subject of said data message, but not allowing any data message published by said data producing processes which does not match the subject of at least one active subscription to be transmitted over said data path.

17. The process of claim 14 wherein data messages transmitted between said data producing processes and said data consuming processes, hereafter referred to as processes, are transmitted as self describing data objects, wherein each self describing data object is comprised of one or more fields each of which is either a primitive class form which stores data or a constructed class form which is comprised of other fields which themselves may be primitive or constructed class forms, each said constructed class form belonging to a class which has a corresponding class definition, said self describing data objects being organized into classes defined by class definitions, each class definition comprising a list of the fields by name and data representation type which are common to all self describing data objects of that class, each self describing data object including both data format information and actual data or field values for each said field, and further comprising the steps of:

automatically converting any self describing data objects to be transmitted from one process to another from the format of the transmitting process to the format necessary for transmission across said data path prior to transmission thereof, and then transmitting said self describing data object through said data path; and

automatically converting any self describing data objects received after transmission through said data path which are bound for either a data consuming process or a data producing process, from the format used to transmit data across said data path to the format used by said receiving process.

18. The process of claim 17 wherein each said conversion step comprises the steps of:

receiving a format conversion call identifying the desired "from" format and the desired "to" format and identifying a specific self describing data object upon which the conversion is to be performed;

accessing a table storing identifiers of particular from-to format conversions;

locating the first field in said self describing data object which is a primitive class form;

using the format information stored in said self describing data object for the primitive field so located, looking up a "from" format in said table corresponding to the format of the primitive class field so located and determining the required "to" format for the conversion specified in said conversion call;

using a table storing pointers to appropriate conversion software routines for the desired from-to format conversion, looking up a from-to format conversion pointer using the "from" and "to" formats just determined and using said pointer to execute an appropriate software routine to perform the desired from-to format conversion;

locating the next field containing a primitive class form in the self describing data object identified in said conversion call and repeating the above steps to convert the data in the next primitive class form from the then existing "from" format to the desired "to" format and repeating this process until all fields containing primitive class forms have been so processed; and

processing every field containing a constructed class form as defined above by searching the constructed class form until the first field is found containing a primitive class form and converting the format thereof using the steps defined above and repeating this process until all fields containing primitive class forms for all levels of nesting of fields containing constructed class forms have been converted to the desired "to" format.

19. The process of claim 14 wherein data transmitted between said data producing processes and said data consuming processes is transmitted as self describing data objects, each self describing data object comprised of one or more fields each of which is either a primitive class form which stores data or a constructed class form which is comprised of other fields which themselves may be primitive or constructed class forms, each said constructed class form belonging to a class which has a corresponding class definition, said self describing data objects being organized into classes defined by class definitions, each class definition comprising a list of the fields by name and data representation type which are common to all self describing data objects of that class, each self describing data object including both data format information and actual data or field values for each said field, for providing the capability for a data consuming or data producing process to obtain data from a particular field of a particular self describing data object generated by another process, comprising the steps of:

receiving in a Get-Field call from a process that desires particular data, said Get-Field call identifying the particular field name of the field of the self describing data object from which the data is to be obtained and the particular one of said self describing data objects from which data is to be obtained;

searching the class definition for said self describing data object from which data is to be obtained to locate said field name identified in said Get-Field call or some synonym thereof;

returning a relative address for said field identified in said Get-Field call, said relative address identifying where in instances of said class of said self describing data object identified in said Get-Field call said field named in said Get-Field call can be found;

accessing said self describing data object identified in said Get-Field call and retrieving the desired data using said relative address of the field in which the desired data is stored, and returning the desired data to the process which requested said data.

20. The process of claim 14 wherein the step of automatically setting up a communications link further comprises the steps of:

sending a subscription request message to a publisher side service discipline process which controls one or more of said computers so as to be coupled to receive data on the subject of said subscription request from a data producing process or processes which control the same said one or more computers having said publisher side service discipline process in execution thereon so as to publish data messages on said subject and with which communication links are to be set up on said subject;

recording all said subscription requests in a subscription table or list;

assigning a channel on said one or more data paths and/or one or more networks to each said subject for which there is an active subscription and recording said channel in said subscription table for each;

when a data message is published by any said data producing process, checking the subject thereof against the subjects of all active subscriptions in said subscription table or list to determine if there are any active subscriptions on the subject;

sending said data message published by any data producing process to all data consuming processes having active subscriptions to the subject of said data by a broadcast communication protocol by dividing said data message into one or more sequential packets suitable for transmission over said data path and/or network, each sequential packet having a header and adding to said header the channel number assigned to said subject and a sequence number indicating where said packet lies in the sequence of packets which, taken together, comprise the data message published by said data publishing process, and calculating error correction bits on data within each said packet and adding said error correction bits so calculated to each said packet;

storing all said packets in a retransmit memory;

sending messages to each data consuming process having an active subscription to the subject of said data informing each data consuming process of the channel on which said data will be broadcast;

broadcasting said data packets on said channel assigned to the subject of said data via said data path and/or network;

receiving said packets at the location of each computer in said computing environment and, at the location of each said computer, comparing the channel number of each said packet so received to the channel numbers recorded in said subscription table of all active subscriptions initiated by any data consuming process in execution at the location of said computer;

if the channel number of the received packets at any computer in said computing environment matches the channel number of any active subscription recorded in said subscription table of a data consuming process in execution on said computer, checking the sequence numbers of all packets received at the location of said computer in said computing environment to determine if all packets comprising the complete data message have been received and using said error correction bits to determine if each received packet has been correctly received, and repeating this process at the location of each said computer, and if the channel number of the received packets at any computer in said computing environment does not match the channel number recorded in said subscription table of a data consuming process in execution on said computer, discarding all said packets whose channel numbers do not match the channel number recorded in said subscription table of a data consuming process in execution on said computer;

if all packets have been successfully received, sending a message back to the process that transmitted the data that all packets have been successfully received, or, if any packet was not received or was not correctly received, sending a message to the process that trans-

mitted the data requesting retransmission of any packets which were not received or which were not correctly received and which cannot be corrected at the receiving computer using said error correction bits;

retransmitting any packets that were not received or which were not correctly received to any computer in said computing environment that transmitted a message indicating one or more packets were not received or were not received correctly;

verifying that the newly transmitted packets have been received and have been correctly received; and

reassembling said data packets into said data message at the location of any computer which received data packets having channel numbers matching the channel number of an active subscription entered by a data consuming process in execution on said computer, and passing said reassembled data message to the data consuming process or processes which requested data on the subject of said data message which is in execution on said computer.

21. The process of claim 14 wherein the step of setting up a communications link further comprises the steps of:

sending a subscription request message to a publisher side service discipline process which controls one or more of said computers so as to be coupled to receive data on the subject of said subscription request from a data producing process or processes which control the same said one or more computers having said publisher side service discipline process in execution thereon so as to publish data on the subject of said subscription request;

recording all said subscription requests in a subscription table and storing in said table the identities of all data consuming processes desiring data on each subject which is the subject of an active subscription;

assigning a channel on said one or more data paths and/or one or more networks to each said subject for which there is an active subscription and recording said channel in said subscription table for each said subscription recorded therein;

when data is published by any said data producing process with which a communication link has been established, checking the subject thereof against the subjects of all active subscriptions in said subscription table to determine how many data consuming processes have active subscriptions on the subject;

comparing the number of said data consuming processes to a predetermined threshold, and determining whether it would be more efficient to transmit the data to all data consuming processes having active subscriptions on said subject using a point-to-point communication protocol by sending the same data multiple times over said one or more data paths and/or one or more networks, one said point to point communication being addressed to each data consuming processes having an active subscription to the subject of said data or by broadcasting said data on a predetermined channel on said one or more data paths and/or one or more networks;

sending said data to all data consuming processes having active subscriptions to the subject of said data by said point-to-point communication protocol if said point-to-point communication protocol is most efficient;

sending said data to all data consuming processes having active subscriptions to the subject of said data by a broadcast communication protocol if said broadcast communication protocol is most efficient;

if said broadcast communication protocol is selected, dividing said data into one or more sequential packets suitable for transmission over a data path; each with a header and adding to said header the channel number assigned to said subject and a sequence number indicating where in the sequence of packets which, taken together, comprise the data message published by said data publishing process, and calculating error correction bits on each packet and adding said error correction bits to each said packet;

storing all said packets in a retransmit memory;

sending messages to each data consuming process having an active subscription to the subject of said data informing each data consuming process of the channel on which said data will be broadcast;

broadcasting on said one or more data paths and/or said one or more networks said data packets via said channel on said one or more data paths and/or one or more networks assigned to the subject of said data;

receiving said packets at each said computer in said computing environment and, at each said computer, comparing the channel number of each said packet to the channel numbers of all active subscriptions listed in said subscription table entered by a data consuming process in execution on said computer;

if the channel number of packets received at a computer in said computing environment matches the channel number of any active subscription listed in said subscription table entered by a data consuming process in execution on said computer, checking the sequence numbers of all packets received to determine if all packets have been received and using said error correction bits to determine if each packet has been correctly received and to correct any errors which can be corrected at the receiving computer using said error correction bits, and if the channel number of packets received at a computer in said computing environment does not match the channel number of any active subscription listed in said subscription table entered by a data consuming process in execution on said computer, discarding all such packets;

sending a message back to the process that transmitted said packets that all packets have been successfully received, or, if any packet was not received or was not correctly received, sending a message to the process that transmitted said packets requesting retransmission of any packets which were not received or which were not correctly received;

retransmitting any packets that were not received or which were not correctly received;

verifying that the newly transmitted packets have been received and have been correctly received; and

reassembling said data packets into a data message and passing the data packets to the data consuming process or processes which requested data on the subject.

22. The process of claim 17 wherein the step of setting up a communications link further comprises the steps of:

sending a subscription request message to a publisher side service discipline process which controls one or more of said computers so as to be coupled to receive data on the subject of said subscription request from a data producing process or processes with which communication links are to be set up on said subject and which control the same said one or more computers having said publisher side service discipline process in execution thereon;

recording all said subscription requests in a subscription table and storing in said table the identities or addresses of all data consuming processes desiring data on each subject which is the subject of an active subscription;

assigning a channel on said one or more data paths and/or one or more networks to each said subject for which there is an active subscription and recording said channel in said subscription table for each said subscription recorded therein;

when data is published by any said data producing process with which a communication link has been established, checking the subject thereof against the subjects of all active subscriptions in said subscription table to determine how many data consuming processes have active subscriptions on the subject;

comparing the number of said data consuming processes to a predetermined threshold, and determining whether it would be more efficient to transmit the data to all data consuming processes having active subscriptions on said subject using a point-to-point communication protocol by sending the same data multiple times over said one or more data paths and/or one or more networks, one said point to point communication being addressed to each data consuming processes having an active subscription to the subject of said data message or by broadcasting said data message on a predetermined channel on said one or more data paths and/or one or more networks and sending a message to all data consuming processes having active subscriptions on said subject to listen for data on the subject of said active subscriptions on said predetermined channel on said one or more data paths and/or one or more networks;

sending said data message to all data consuming processes having active subscriptions to the subject of said data by said point-to-point communication protocol if said point-to-point communication protocol is most efficient;

sending said data message to all data consuming processes having active subscriptions to the subject of said data message by a broadcast communication protocol if said broadcast communication protocol is most efficient;

if said broadcast communication protocol is selected, dividing said data message into one or more sequential packets suitable for transmission over a data path, each with a header and adding to said header the channel number assigned to said subject and a sequence number indicating where in the sequence of packets which, taken together, comprise the data published by said data publishing process, and calculating error correction bits and adding said error correction bits to each said packet;

storing all said packets in a retransmit memory;

broadcasting on said one or more data paths and/or said one or more networks said data packets on said channel on said one or more data paths and/or one or more networks assigned to the subject of said data;

receiving said packets at each said computer in said computing environment and, at each said computer, comparing the channel number to the channel numbers of all active subscriptions listed in said subscription table entered by a data consuming process in execution on said computer;

if the channel number of packets received at a computer in said computing environment of a packet matches the

channel number of any active subscription, listed in said subscription table entered by a data consuming process in execution on said computer, checking the sequence numbers of all packets received to determine if all packets have been received and using said error correction bits to determine if each packet has been correctly received and to correct any errors which can be corrected at the receiving computer using said error correction bits, and if the channel number of packets received at a computer in said computing environment does not match the channel number of any active subscription listed in said subscription table entered by a data consuming process in execution on said computer, discarding all such packets;

sending a message back to the data producing process that all packets have been successfully received, or, if any packet was not received or was not correctly received, sending a message to the data producing process that published the data requesting retransmission of any packets which were not received or which were not correctly received;

retransmitting any packets that were not received or which were not correctly received;

verifying that the newly transmitted packets have been received and have been correctly received; and

reassembling the data packets into the original data message and passing the data message to the data consuming process or processes which requested data on the subject.

23. An apparatus for obtaining data requested by one or more data consuming processes in execution on one or more computers from one or more service instances in execution on one or more hosts or server computers, comprising:

one or more networks or other data paths for transporting data between processes running at various locations or addresses on said network(s), said transport of data being carried out on one or more appropriate communication paths through said network(s) or other data path(s), and according to one or more appropriate communication mechanisms or transport protocols;

one or more host and/or server computers coupled to said network(s), each having one or more network addresses, each host and/or server computer having an operating system and one or more other process(es), data consuming process(es) or service instance(s) in execution thereon, each of said operating system(s), process(es), data consuming process(es) or service instance(s) being programmed in any selected programming language, each said host and/or server computer having any selected machine architecture or type including any appropriate machine instruction set and any appropriate data representation format or type, each of said operating system(s), process(es) and/or service instance(s) having one or more appropriate communication, access and/or invocation protocol(s);

one or more computers having in execution thereon one or more data location and access programs which control processing by said one or more computers, said data location and access programs being coupled to said one or more data consuming process(es) and said one or more other process(es) and/or service instance(s), for receiving one or more requests for desired data from said one or more data consuming processes, said request defining the identity of the desired data by an identifier comprising one or more parts, said desired data having certain access requirements including but

not limited to the particular communication path(s) through said network(s) or other data path(s) to said desired data, the transport protocol or protocols along said communication path(s) through said network(s) or other data path(s) to said desired data, the network address(es) of the host(s) and/or server computer(s) on which the process(es) publishing the desired data is or were in execution, the machine architecture(s) or type(s) or operating system(s) or the data representation format or type of the host(s) and/or server computer(s) upon which the process(es) publishing the desired data is or were in execution, the particular process(es) and/or service instance(s) or the programming language(s) of the particular process(es) and/or service instance(s) which is or were publishing said desired data, and the communication, access or invocation protocol(s) which must be invoked to communicate, access or invoke said process(es) or service instance(s) which is or were publishing said desired data or the transport protocol(s) or operating system or systems in execution on any of said host or server computers involved in accessing and transporting said desired data from the process(es) which publish or distribute said desired data to the data consuming process which requested said data, all of said access requirements uniquely defining the attributes of a communication mechanism between said data consuming process(es) desiring said data and said desired data itself, said request for said desired data and said data consuming process which made said request being independent of one or more of said access requirements or attributes including at least the identity and/or location of said one or more particular process(es) and/or service instance(s) that publish or distribute said requested data and said communication, access or invocation protocol or protocols needed to communicate with said one or more process(es) and/or service instance(s) that publish or distribute said requested data as well as the location or address of the one or more computer(s) being controlled by said one or more processes and/or service instance(s) that publish or distribute said requested data, said independence meaning that the data consuming process which requested said desired data need include no program, code or data the purpose of which is to satisfy any of said access requirements, said one or more computers processing of which is controlled by said one or more data location and access programs being controlled thereby so as to automatically satisfy all said access requirements so as to establish a communication path(s) to said desired data, access said data and return said data to said data consuming process(es) that requested said data, and wherein said one or more process(es) and/or service instance(s) that publish or distribute said requested data also do not include any program, code or data the purpose of which is to satisfy any said access requirement, other than to output the subject itself of the data being published, or the purpose of which is to direct where said data is to be transmitted or to assist in any other way in satisfying any other of said access requirements.

24. The apparatus of claim 23 wherein said one or more data location and access programs include one or more data format decoupling programs coupled to said one or more computers having in execution thereon said one or more data location and access programs, said one or more data format decoupling programs for providing data representation independence such that data may be effectively obtained by said

93

one or more data consuming process(es) which requested said data regardless of the particular data representation format(s) or type(s) used by said one or more data consuming process(es) and whatever process(es) or service instance(s) which publish the requested data.

25. The apparatus of claim 24 wherein said one or more data format decoupling programs further comprise means coupled to said one or more data consuming process(es) and said one or more process(es) and/or service instance(s) which publish the requested data for creating, manipulating, storing and transmitting self-describing data objects which include not only data but also data representation format information within each instance of a self describing data object.

26. The apparatus of claim 23 wherein said one or more data location and access programs include one or more service discipline programs for encapsulating appropriate software routines for communicating with one or more of said process(es) and/or service instance(s) which publish said requested data in the communication, access and/or invocation protocol(s) native thereto.

27. The apparatus of claim 24 wherein said one or more computers having in execution thereon said one or more data location and access programs for receiving said request(s) for desired data also are programmed to implement one or more service discipline means for encapsulating and executing appropriate software routines for controlling one or more computers so as to communicate with one or more of said process(es) and/or service instance(s) which publish said requested data using the communication, access and/or invocation protocol(s) native thereto, and wherein said self-describing data objects are organized into classes, each of which has common attributes and a unique class identifier, and wherein each self describing data object has one or more fields, each field being either primitive in that the field stores data or constructed in that the field stores data which is the class identifier of another class of self describing data objects and the data from an instance from said other class.

28. The apparatus of claim 23 wherein said one or more data location and access programs includes Subject-Addressed Subscription Service means for controlling said one or more computers to implement a programmatic interface to said one or more data consuming and data publishing processes, said programmatic interface implementing a subscribe function which can be invoked by a data consuming process by outputting a a subscription request requesting data by identifier only, said data and all updates thereto published under said particular identifier by one or more of said process(es) and/or service instance(s), and for automatically establishing a communication path between at least one of said data consuming processes which requested said data by identifier only and at least one of said processes and/or service instances which publish said requested data having said identifier until said subscription request is canceled, said programmatic interface also for controlling one or more computers having one or more service instances in execution thereon such that service instance(es) can publish data simply by invoking a publish function of said programmatic interface and outputting the data and the identifier of the data, and wherein said computer(s) controlled by said programmatic interface will use the identifier of said published data to automatically route the data to only those computers having in execution thereon a data consuming process having an open subscription to data having said identifier and to no other computer, said programmatic interface also controlling one or more computers such that when said process(es) and/or service instance(s) publish

94

updates to said requested data with an identifier, said identifier is used to transmit said updates automatically to said one or more data consuming processes which requested said data for which said communication path(s) has or have been established and to only those computers having in execution thereon data consuming processes having open subscriptions to data having said identifier until said subscription is canceled.

29. The apparatus of claim 25 wherein said self-describing data objects are organized into classes, each of which has common attributes and a unique class identifier, and wherein each self-describing data object has one or more fields, each field being either primitive in that the field stores data or constructed in that the field stores data which comprises the class identifier of another class of self-describing data objects and the data from an instance from said other class.

30. The apparatus of claim 28 wherein said one or more data location and access programs further further comprises means, coupled to said one or more data consuming processes which requested said data and said one or more process(es) and/or service instance(s) which publish said requested data, for creating, manipulating, storing and transmitting said requested data as self-describing data objects which include not only data but also said data representation format or type information within each instance of a data object.

31. The apparatus of claim 30 wherein said self-describing data objects are organized into classes, each of which has common attributes and a unique class identifier, and wherein each self-describing data object has one or more fields, each field being either primitive in that the field stores data or constructed in that the field stores data which comprises the class identifier of another class of self-describing data objects and the data from an instance from said other class.

32. The apparatus of claim 28 wherein said one or more data location and access programs include one or more service discipline means, each for controlling one or more of said computers for communicating with one or more of said processes and/or service instances which publish said requested data in the communication, access or invocation protocol(s) native thereto.

33. The apparatus of claim 31 wherein said one or more data location and access programs include one or more programs which control said one or more computers to implement one or more service discipline means, each for communicating with one or more of said processes and/or service instances which publish said requested data in the communication, access or invocation protocol(s) native thereto.

34. The apparatus of claim 28 wherein said one or more computers having in execution thereon said one or more data location and access programs also are programmed to also implement means for establishing said communication path by listening for data messages from any source arriving at one or more network pod address(es) associated with said one or more data consuming processes which requested said data and filtering out all data messages other than data messages which have said identifier associated with said requested data, and passing all said messages having said identifier of said requested data to any said one or more computers having in execution thereon said one or more data consuming process(es) which issued said request(s) for said data and for which said communication path has been established.

35. The apparatus of claim 28 wherein said one or more data location and access programs include one or more programs to control one or more of said computers so as to

implement subscription registration means for establishing said communication path by sending a subscription registration message to register said subscription for said desired data with the one or more computers having in execution thereon said one or more data location and access programs which are coupled to said process(es) and/or service instances which publish said requested data, said subscription registration message including the identifier of the requested data, and wherein said at least one process and/or service instance which publishes said requested data is coupled to said one or more computers implementing said subscription registration means and wherein said one or more computers having in execution thereon said one or more data location and access programs is also programmed to implement means for transmitting data published by said at least one process and/or service instance which has an identifier which matches the identifier for requested data which is the subject of any active subscription registration to all the data consuming process(es) which registered a subscription to data having said identifier.

36. The apparatus of claim 32 wherein each said computer programmed to implement said one or more service discipline means is also programmed to implement means for failure recovery so as to be able to maintain the flow of data to said one or more data consuming process(es) which requested said data despite failure of said communication path, host and/or server computer(s) or process(es) and/or service instance(s) which is publishing said requested data.

37. The apparatus of claim 28 wherein said Subject-Addressed Subscription Service means includes service discipline means for controlling said one or more computers to carry out communication of data over said communication path to fulfill said subscription request using a communication protocol which is appropriate for the one or more processes and/or service instances which are supplying said data, and wherein said one or more data location and access programs include one or more programs for controlling said one or more computers to implement one or more protocol engines, said protocol engines for establishing said communication path(s) using the appropriate network or communication path transport protocols for the communication path through which said requested data is obtained and for interfacing said service discipline means to said network or communication path transport protocols.

38. The apparatus of claim 25 wherein said one or more data location and access programs include means for implementing one or more service disciplines for controlling said one or more computers to communicate with service instances in the communication, access or invocation protocol native thereto, and wherein said one or more data location and access programs coupled to said one or more other process(es) and/or service instance(s) and said one or more data consuming process(es) include protocol engine means for controlling one or more computers to receive requests from said service disciplines to establish a communication path through which said requested data may be obtained, and assisting said one or more computers controlled by said service disciplines so as to establish said communication path(s) by interfacing said one or more computers controlled by said one or more service disciplines to the appropriate network or communication path transport protocols for the communication path through which said requested data is to be obtained.

39. The apparatus of claim 37 wherein said protocol engines include means for controlling said one or more computers to carry out a reliable broadcast communication protocol.

40. The apparatus of claim 39 wherein each said protocol engine controls said one or more computers so as to have different fault tolerance characteristics not already implemented by said transport protocols.

41. The apparatus of claim 38 wherein said one or more computers controlled by said protocol engine means is controlled so as to carry out a reliable broadcast communication protocol if the number of subscribers to data having a particular identifier is greater than a programmable number and is controlled to carry out a point-to-point communication protocol if the number of subscribers is less than said programmable number.

42. The apparatus of claim 41 wherein each said a protocol engine controls said one or more computers so as to have different fault tolerance characteristics.

43. The apparatus of claim 23 or 28 wherein said requested data is transported over said communication path using one or more data messages, and wherein said one or more data location and access programs includes means for transporting said requested data over said communication path using a reliable communication protocol which insures that all data messages were received without error.

44. The apparatus of claim 23 or 28 wherein said one or more data location and access programs includes reliable broadcast transmit means coupled to said process(es) and/or service instance(s) which publish said requested data, said reliable broadcast transmit means for dividing data messages into packets for transmission on said network and adding sequence numbers to said packets, and for calculating error correction bits for each packet and adding said error correction bits to said packets, and for temporarily storing packets to be transmitted over said network in a retransmit memory, and for broadcasting said packets over said network or communication path, and wherein said one or more data location and access programs further comprise receive means coupled to said data consuming processes which requested said data for listening to a network address associated with at least one of said data consuming processes which requested said data and selecting incoming packets which comprise a complete data message of said requested data and checking the sequence numbers of incoming packets to insure that all data messages comprising said requested data have been received, and for using said error correction bits of each packet to detect and correct errors in said received packets, and, if any data message was not received or contains errors not correctable using said error correction bits, for sending a message back to said reliable broadcast transmit means requesting retransmission from said retransmit memory of any data messages not received or not received correctly.

45. The apparatus of claim 23 or 28 wherein said one or more data location and access programs further comprises an intelligent multicast program for controlling one or more computers so as to determine how many data consuming processes have open subscriptions to data with a particular identifier and for controlling said one or more computers to broadcast data messages having said identifier when the number of data consuming processes requesting said data is greater than or equal to a predetermined number, and for sending said data messages directly to the data consuming process or processes which requested said data by a point-to-point communication protocol when the number of said data consuming processes which requested said data is below said predetermined number, said predetermined number selected such that the most efficient means of transporting said data is used in terms of consumption of network or communication path bandwidth and consumption of computing resources.

46. The apparatus of claims 23 or 28 wherein said one or more computers having in execution thereon said one or more data location and access programs are controlled thereby so as to implement means for checking the identifier of said requested data and the identity or identities of said data consuming processes which requested said data against a list of authorized identifiers associated with each of said one or more data consuming processes which requested said data, and for blocking access by any data consuming process to any requested data having an identifier not on the list of authorized identifiers for said data consuming process.

47. The apparatus of claim 23 or 28 wherein said one or more computers having in execution thereon one or more data location and access programs are also programmed by said one or more data location and access programs to implement means for switching communication paths through said network or other data communication path upon failure of the selected communication path or other interruptions in the flow of data so as to maintain the flow of data to all data consuming processes having open subscriptions.

48. An apparatus for use in a distributed computing environment, comprising:

one or more computers coupled by one or more data paths, said one or more computers having in execution thereon one or more application processes controlling processing of said one or more computers, and wherein at least one of said application processes is a data consumer process needing data on one or more subjects and capable of outputting a subscription request for each subject for which data is desired, each said subscription request including a subject for which data is requested but wherein no data consumer process needs to include any routine which controls one more of said computers so as to output any information, address data or address related data indicating where said data may be found in said distributed computing environment other than outputting the subscription request and the subject thereof, and wherein at least one of said application processes is a data publisher process which outputs data messages on various subjects but which does not need to include any routine the purpose of which is to output any address or address related data, other than the subject itself, which indicates where said data messages are to be transmitted in said distributed computing environment or in any other way assist in the distribution of said data;

one or more computers processing of which is controlled by one or more directory services programs or routines so as to, when given a subject, access and search a data file, said data file storing data records mapping data publisher processes to subjects, each said data record containing a subject and identity and/or address data specifying, directly or indirectly, the address in said distributed computing environment of one or more sources of data messages on said subject, each said source obtaining said data messages on said subject from one or more data publisher processes in said distributed computing environment which publish data on said subject, and wherein each source on any particular subject comprises one or more computers controlled by one or more computer programs so as to receive messages from a data publisher process on the subject corresponding to said source and re-transmit said data messages on the subject to only those data consumer processes which have open subscriptions for data messages on the subject, said one or more directory services programs or routines for controlling said

one or more computers so as to return any data record in said data file identifying and/or giving the address in said distributed computing environment of one or more sources which can output data on said subject that was originally passed to said one or more computers on which said one or more directory services programs or routines are in execution;

one or more computers controlled by one or more mapping programs or routines so as to receive each said subscription request from a computer on which a data consumer process is in execution, said subscription request identifying only a particular subject for which a subscription is requested, said one or more mapping programs or routines for controlling said one or more computers so as to map the subject of said subscription request directly or indirectly to the identity and/or address in said distributed computing environment of one or more sources which can output data messages on said subject, said mapping carried out by said one or more computers controlled by said one or more mapping programs or routines by passing the subject of said subscription request to said one or more computers controlled by said at least one directory services program or routine so as to cause said one or more computers controlled by said one or more directory services programs or routines to search said data file for a data record(s) identifying and/or giving the address in said distributed computing environment of one or more sources which output data on said subject and controlling said one or more computers on which said one or more mapping programs or routines are in execution to receive said data records resulting from said search, said one or more mapping programs for controlling said one or more computers so as to output at least said identity or address data in a link request requesting the establishment of a subscription communication link with said one or more sources identified in said data records resulting from said search;

one or more computers controlled by one or more subscription registration and communication programs or routines so as to receive said identity and/or address data in said link request from said one or more computers controlled by said one or more mapping programs or routines, said link request corresponding to at least one subject and a corresponding subscription request, said subscription registration and communication programs or routines for controlling said one or more computers so as to automatically establish a communication link with at least one source which outputs data on the subject to which said link request pertains via at least one said data path and automatically send a subscription registration message to said at least one source identified in said link request so as to notify said source of an open subscription on said subject, each said subscription registration message directly or indirectly identifying and/or giving the address in said distributed computing environment of the data consumer process which issued said subscription request on said subject through the computer on which said data consumer process is in execution or some other process which controls one or more computers to receive data messages on the subject of said subscription request on behalf of said data consumer process which requested the data and pass the data messages only to said one or more computers controlled by said data consumer process which requested said data, said one or more subscription registration and

communication programs also for controlling said one or more computers so as to receive data messages on said subject whenever said data messages on said subject are output by said source with which said subscription communication link has been established and pass said data messages to said one or more computers being controlled by said data consumer process which requested data on the subject until said subscription is canceled;

one or more computers under control by one or more data distribution programs or routines so as to implement said one or more sources which output data on the subject of each said subscription request, said sources being selectively coupled to said one or more computers under control of said data consumer processes and selectively coupled to said one or more computers under control of said data publisher processes, said one or more computers under control of said one or more data distribution programs or routines for receiving data messages published by said one or more data publisher processes on one or more subjects, and for receiving said subscription registration messages requesting subscriptions on one or more subjects and for automatically sending each said data message over said subscription communication link only to computers having in execution thereon data consumer processes having active subscriptions to the subject of said data message, said data distribution programs or routines controlling said one or more computers to so distribute published data without receiving any data from the data publisher process, other than the subject itself of each data message, indicating where each said data message on any subject is to be sent, and for automatically continuing to transmit each new data message received from a computer controlled by a data publisher process on any subject to only data consumer processes having an active subscription to the subject of each said new data message until said subscription is canceled, each said data distribution program(s) or routine(s) containing no routines, computer program, subroutine or other computer programming code the purpose of which is to receive any data from said data publisher process, other than the subject of a data message itself, which controls where in said distributed computing environment any said data message on any subject is to be sent.

49. A process for communicating data between at least one publishing process and at least one consumer process in execution on one or more computers in a distributed computing system, said publishing and consumer processes coupled by a data path, comprising:

storing data encoding a subject space and the subjects therein in a memory of one or more of said computers, and storing in said memory data mapping the identity and/or the address in said distributed computing system of one or more source computer processes for execution on one or more of said computers in said distributed computing system which can supply data on a subject for every subject in said subject space;

receiving at one or more computer processes in execution on one or more of said computers a subscription request from at least one consumer process, said subscription request specifying only the subject for which data is requested but not where data on said subject can be found in said distributed computing system and specifying a manner of returning data on the requested subject to said consumer process;

in one or more computer processes in execution on one or more of said computers, using the subject of said subscription request to automatically look up in said stored data encoding said subject space and the subjects therein, the identity and/or address in said distributed computing system of one or more source computer processes which can supply data on the subject specified in said subscription request, and automatically sending a subscription registration message to said source computer process specifying the subject for a subscription and specifying the manner of returning data on the specified subject to the consumer process which requested said data, said subscription registration message being sent to one or more subscription registration processes in execution on said one or more computers the function of which is to keep one or more subscription lists listing all active subscriptions on each subject;

receiving at said one or more source computer processes in execution on one or more of said computers one or more data messages from at least one publisher process which specify a subject and include data to be published on said subject but which do not specify where the data on said subject is to be sent within said distributed computing system other than by specifying the subject itself, and comparing the subject of each said data message to said one or more subscription lists kept by said one or more subscription registration processes and, if one or more active subscriptions exist for the subject of any particular data message, automatically sending all data messages on the subject and any subsequent data messages on the same subject to only those computers which have consumer processes in execution thereon having open subscriptions on the subject of said data messages, and for continuing to send each data message pertaining to a subject to only those computers having in execution thereon a consumer process having an active subscription for said subject until the subscription on said subject is canceled.

50. A distributed computing apparatus, comprising:

a plurality of 30 or more computers processing of which is controlled by a plurality of 30 or more programs or subroutines, said programs or subroutines in execution on one or more of said plurality of 30 or more computers, hereafter referred to as processes, said plurality of processes and said plurality of computers coupled by one or more data paths;

and wherein at least one of said computers has in execution thereon at least one of said processes which is a subscriber process which requests data on any subject in a hierarchically organized subject space having 30 or more subjects by issuing a subscription request requesting that a subscription be opened and naming the subject of the requested data and including data indicating how data on said subject is to be sent to said subscriber process, where no subscription request includes any data, other than the subject itself, indicating the identity or locations in said distributed computing environment of any publisher process which outputs data messages on the subject named in said subscription request;

and wherein at least one of said processes is a publisher process which publishes data messages on one or more subjects and which does not include any routine, computer code or subroutine the purpose of which is to receive any data, address or address related data the

purpose of which is to control directly or indirectly where data being published by said publisher process is to be sent in said distributed computing environment or to send any data, address or address related data with said data messages being published, other than the subjects themselves of the data messages, which indicates or controls directly or indirectly where in said distributed computing apparatus to send any said data message;

one or more computers under control of one or more subject-based addressing programs or subroutines said one or more computers under control of said one or more subject-based addressing programs or subroutines being coupled to each said publisher process and each said subscriber process via said one or more data paths, said one or more subject-based addressing programs comprising one or more subscription handling routines for controlling execution on one or more of said computers so as to receive said subscription requests from said one or more subscriber processes and for keeping a list of active subscriptions of subjects for which there have been subscription requests made by one or more subscriber processes, and for automatically locating, for each active subscription, one or more computers coupled to or controlled by a process which can supply data on the subject of said subscription request, said one or more computers having their processing controlled by one or more programs which cause said one or more computers to supply data on the subject of each said subscription request, and said one or more computers under control of one or more subject-based addressing programs or subroutines for automatically establishing a data communication path for each said subscription request between the subscriber process which issued said subscription request and said one or more computers coupled to or controlled by said process that can supply data on the subject of said subscription request and automatically registering a subscription on the subject of said subscription request on said one or more computers coupled to or controlled by said process that can supply data on the subject so as to start a flow of data on said subject over said data communication path previously established, and said one or more subject-based addressing programs or subroutines for controlling said one or more computers so as to pass each data message received from said one or more computers coupled to or controlled by said process which can supply data on the subject of said subscription request only to computers under control of said one or more subscriber processes having an active subscription on the subject of each said data message.

51. The apparatus of claim 50 wherein said one or more computers controlled by said one or more subject-based addressing programs further comprise one or more computers controlled by one or more published message routing processes so as to receive data messages published by said one or more computers coupled to or controlled by said process which can supply data on the subject of each said subscription request, and, without receiving any data, command or request from any said publisher process controlling, assisting in controlling or indicating where each said data message is to be sent, for automatically and continuously sending each data message published on any said subject only to computers controlled by subscriber processes that have active subscriptions on the subject of said data message using the one or more data communication paths that have

been opened for that subject and, wherein, for each said subscription request, said one or more computers controlled by one or more published message routing processes and said one or more subject-based addressing programs or subroutines automatically carry out one or more communication protocols and any necessary data format conversion operations that enable the computer controlled by a subscriber process which issued said subscription request on a subject to effectively communicate with the one or more computers coupled to or controlled by said process which can supply data on the subject of said subscription request.

52. A process for communicating data between software processes operating in one or more computers coupled by a data communication path, comprising:

storing data encoding a subject space and the subjects therein in a memory of one or more of said computers, and storing in said memory data mapping the identity and/or the address of one or more source processes for execution on one or more of said computers which can supply data on a subject for every subject in said subject space;

receiving a subscription request for information on a particular subject from a requesting software process and locating at least one suitable said source process for data on said subject by searching said data stored in said computer memory encoding said subject space using the subject of said subscription request as a search criteria; and

automatically satisfying all access requirements including invoking and carrying out all necessary communication protocols and data format and representation conversions needed to set up a communication channel through said data communication path between said source process and said requesting software process such that a subscription is set up whereby current data on the requested subject and subsequent data updates on the requested subject flows through said communication channel only to computers being controlled by the requesting software process(es) and are converted automatically to the data format and representation used by said requesting software process(es) until said subscription request is canceled, all said access requirements being satisfied, communication protocols invoked and data format conversions being carried out automatically without the aid of any data from or processing by said source process or requesting process such that said requesting software process need only name the subject of the desired data and said source process need only output data on the requested subject and name the subject thereof with the necessary processing needed to get that data to the requesting software process and the configuration and/or operation of the data communication path being transparent to said source process.

53. A process for communicating data between software processes controlling processing of one or more computers coupled by a data communication path, one or more of said software processes being a consumer process that needs data and one or more of said software processes being a publisher process which publishes data on a subject, said publisher process, said consumer process and said data communication path having access requirements, comprising:

storing data encoding a subject space and the subjects therein in a memory of one or more of said computers, and storing in said memory data mapping the identity and/or the address of one or more publisher processes for execution on one or more of said computers which

can supply data on a subject for every subject in said subject space;

receiving a subscription request for information on a particular subject from one or more consumer software processes where no consumer process needs to output any data or address information which specifies where the desired data can be found other than the subject itself, and locating at least one suitable publisher process which can publish data on said subject by searching said data stored in said computer memory encoding said subject space using the subject of said subscription request as a search criteria; and

automatically satisfying all said access requirements including invoking and carrying out all necessary communication protocols needed to set up a communication channel through said data communication path between said publisher process and only said computers processing of which is controlled by consumer software processes having open subscriptions on the subject of said subscription request such that a subscription is set up whereby current data on the requested subject and subsequent data updates on the requested subject flow through said communication channel and are routed automatically to only computers processing of which is controlled by consumer software processes which requested data on the specified subject until said subscription request is canceled, all said access requirements being satisfied and communication protocols being invoked automatically without the aid of any data from or processing by said publisher process, other than by output of the subject itself, such that said publisher process need only output data on the requested subject and name the subject thereof with no need for said publisher process to have therein any software routine which knows the locations of consumer processes having open subscriptions to the subject or which is capable of finding the address of any said consumer process having an open subscription on the subject of the published data, with all the necessary processing needed to get the data published by said publisher process to all consumer software processes having open subscriptions on the subject of said published data by nonbroadcast communication processes being automatically performed by middleman software processes other than the publisher process or the consumer process or processes and wherein the configuration and operation and communication protocols of the data communication path and said middleman software processes are transparent to said publisher process and said consumer process or processes.

54. An apparatus comprising:

one or more computers having one or more software processes in execution thereon controlling operations of said one or more computers, at least one of said software processes being a consumer process that needs data on a subject and at least one of said software processes being a publisher process which publishes data on one or more subjects,

one or more data paths coupled to said one or more computers and capable of carrying data between said consumer and publisher processes in execution on said one or more computers;

and wherein said one or more computers have intermediary software program(s) in execution thereon for controlling one or more of said computers to implement decoupled, subject based addressing whereby a con-

sumer process can obtain data pertaining to a subject simply by entering a subscription request by invoking a subscribe function of an application programmatic interface, hereafter referred to as an API, implemented by said intermediary software program(s), said subscription request only naming the subject of the desired data and identifying a method for getting data on the requested subject back to the consumer process, and said intermediary software program(s) for controlling one or more computers to take the subject of said subscription request and automatically map said subject to the identity and/or address of one or more publisher processes which is/are capable of supplying data on said subject and wherein said intermediary software program(s) control one or more computers to automatically set up a data communication path for each said subscription request between said one or more publisher processes which publish data on the subject of said subscription request and only those computers being controlled by consumer processes having an open subscription to the subject of said subscription request, and wherein said API implemented by said intermediary software program(s) provides a publish function which can be invoked by a data publisher process when data on a subject is to be published thereby allowing a publisher process to have published data distributed to all consumer processes having open subscriptions to the subject of said published data simply by invoking said publish function, publishing said data and naming the subject thereof, and, when said publish function is invoked, said intermediary software program(s) receives the published data and automatically distributes said data only to computers controlled by consumer processes that have active subscriptions to the subject of the published data.

55. An apparatus comprising:

one or more computers having one or more software processes in execution thereon controlling operations of said one or more computers, at least one of said software processes being a consumer process that needs data on a subject and at least one of said software processes being a publisher process which publishes data on one or more subjects;

one or more data paths coupled to said one or more computers and capable of carrying data between said consumer and publisher processes in execution on said one or more computers;

and wherein said one or more computers have one or more intermediary software means in execution thereon for controlling one or more of said computers to implement decoupled, subject based addressing, said intermediary software means including one or more means for controlling said one or more computers to provide an application programmatic interface, hereafter referred to as an API, to each said consumer process, whereby each said consumer process can obtain data pertaining to a subject simply by entering a subscription request in the form of invoking a subscribe function of said API implemented by said intermediary software means, said subscription request naming the subject, and wherein said intermediary software means includes one or more mapping means for controlling said one or more computers to automatically map the subject of said subscription request to the identity and/or address of one or more publisher processes which is/are capable of supplying data on said subject, and wherein said intermediary software means includes one or more means for

controlling said one or more computers to automatically establish a data communication path between all consumer processes having open subscriptions on a subject and one or more publisher processes which can publish data on the subject, and wherein said intermediary software means includes one or more means for implementing a publisher side API to provide a facility such that a data publisher process can invoke a publish function when data on a subject is to be published, and wherein said intermediary software means is also for controlling one or more computers to receive the published data when said publish function of said publisher side API is invoked and distribute said data automatically to only computers controlled by consumer processes that have active subscriptions to the subject of the published data and without the need for said publisher process to include any data or software routines that contains or output the addresses of any consumer process or which is designed to designate to which consumer processes said data is to be distributed or which is designed to locate any consumer process which has an open subscription to data on the subject of said published data or any other subject.

56. An apparatus for facilitating communication of data in a distributed system between two or more computer programs in execution on the same or different computers coupled by a data exchange medium, comprising:

- a network comprised of at least one data transfer path, said network coupling said one or more computers by one or more data transfer paths;
- at least one computer execution of which is controlled by one or more application computer programs, said application computer program controlling said one or more computers by issuing a subscription request for data on a subject named in said subscription request;
- at least one computer execution of which is controlled by one or more data publishing computer programs which may or may not be the same as said application computer program, said data publishing computer program controlling execution on at least one said computer so as to output data on at least one subject;
- one or more computers controlled by one or more subject based addressing programs so as to logically decouple said one or more application computer programs from said one or more data publishing computer programs by providing an application programmatic interface to said one or more application computer programs such that said one or more application computer programs need not have any routines therein to locate or communicate with any other computer or computer program to obtain data on a subject other than to issue said subscription request through said application programmatic interface to said one or more computers controlled by said subject based addressing program and said one or more

computers controlled by said subject based addressing program will automatically locate a computer controlled by a data publishing computer program which publishes data on said subject and will obtain data on said subject and pass said subject to said application computer program which issued said subscription request thereby insulating said application computer programs from obsolescence when changes or substitutions occur in the one or more data publishing computer programs which controls said one or more computers to publish data requested by said one or more application computer programs, said subject based addressing programs also for controlling said one or more computers such that data published by one or more computers controlled by one or more data publishing computer programs that is on a subject for which there is no active subscription is filtered out by the one or more computers executing said one or more data publishing computer programs such that said data is never transmitted on said network, said subject based addressing programs also for controlling said one or more computers such that data published by said one or more computers executing said one or more data publishing computer programs that is on one or more subjects for which there are active subscriptions is transmitted only to computers controlled by said one or more application computer programs that issued subscription requests on the subject of said data, said subject based addressing programs also for controlling said one or more computers such that an application programmatic interface is presented to one or more said data publishing programs such that said one or more computers executing said data publishing programs can transmit data published on a subject for which there is an active subscription only to those computers execution of which is controlled by one or more application computer programs which requested said data by outputting said data and the subject thereof through said application programmatic interface to said one or more computers controlled by said one or more subject based addressing computer programs such that said one or more data publishing need not include any routines or computer program instructions designed to locate or communicate with any computer controlled by one or more application computer programs other than to output data and the subject thereof through said application programmatic interface thereby insulating said data publishing computer programs from obsolescence when changes or substitutions occur in the one or more application computer programs which controls said one or more computers to request data published by said one or more data publishing computer programs.

* * * * *



US005950201A

United States Patent [19]

Van Huben et al.

[11] Patent Number: **5,950,201**[45] Date of Patent: **Sep. 7, 1999**

[54] **COMPUTERIZED DESIGN AUTOMATION METHOD USING A SINGLE LOGICAL PFVL PARADIGM**

[75] Inventors: **Gary Alan Van Huben; Joseph Lawrence Mueller**, both of Poughkeepsie, N.Y.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **08/759,692**

[22] Filed: **Dec. 6, 1996**

[51] Int. Cl.⁶ **G06F 15/173**

[52] U.S. Cl. **707/10; 707/4; 707/8; 707/102; 707/203; 364/468.02; 395/200.31**

[58] Field of Search **707/10, 102, 8, 707/203, 4; 364/468.02; 395/200.31**

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,191,534	3/1993	Orr et al.	364/468.12
5,201,047	4/1993	Maki et al.	707/4
5,216,612	6/1993	Cornett et al.	364/468.02
5,317,729	5/1994	Mukherjee et al.	707/3
5,321,605	6/1994	Chapman et al.	705/7
5,333,312	7/1994	Wang	707/10
5,333,315	7/1994	Saether et al.	707/1
5,333,316	7/1994	Champagne et al.	707/8
5,418,949	5/1995	Suzuki	707/205
5,463,555	10/1995	Ward et al.	364/468.02
5,530,857	6/1996	Gimza	707/10
5,586,039	12/1996	Hirsch et al.	364/468.01

OTHER PUBLICATIONS

Norrie et al. "Flexible Enterprise through Coordination Repositories", Institute for Information Systems, The 11th ISPE/IE/IFAC International Conference on CAD/CAM Robotics and Factories on the Future '95, pp. 135-140, 1995.

Tony-Clifford-Winters, "The Repository-Understanding IBM", The relational Journal, Issue No. 12, pp. 5-10, Jan. 1991.

Oliver Tegel, "Integrating Human Knowledge Into The Product Development Process" published in Proceedings of ASME Database Symposium Eng-Data Mgmt, Integrating the Engineering Enterprise ASME Database Symposium 1994, ASCE NY USA, pp. 93-100.

"Beyond EDA" published in Electronic Business, vol. 19, No. 6, Jun. 1993 pp. 42-46, 48.

Primary Examiner—Paul V. Kulik

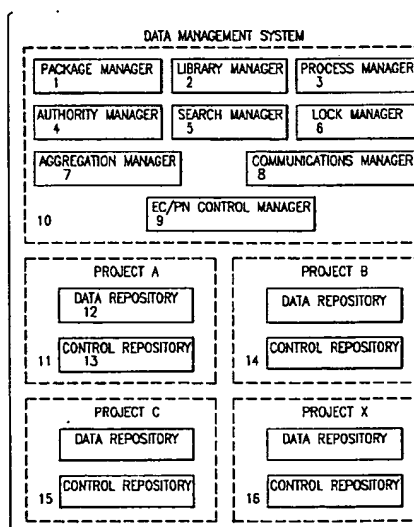
Assistant Examiner—Jean R. Homere

Attorney, Agent, or Firm—Lynn L. Augspurger

[57] **ABSTRACT**

A design control system suitable for use in connection with the design of integrated circuits and other elements of manufacture having many parts which need to be developed in a concurrent engineering environment with inputs provided by users and or systems which may be located anywhere in the world providing a set of control information for coordinating movement of the design information through development and to release while providing dynamic tracking of the status of elements of the bills of materials in an integrated and coordinated activity control system utilizing a repository which can be implemented in the form of a database (relational, object oriented, etc.) or using a flat file system. Once a model is created and/or identified by control information design libraries hold the actual pieces of the design under control of the system without limit to the number of libraries, and providing for tracking and hierarchical designs which are allowed to traverse through multiple libraries. Data Managers become part of the design team, and libraries are programmable to meet the needs of the design group they service.

26 Claims, 26 Drawing Sheets



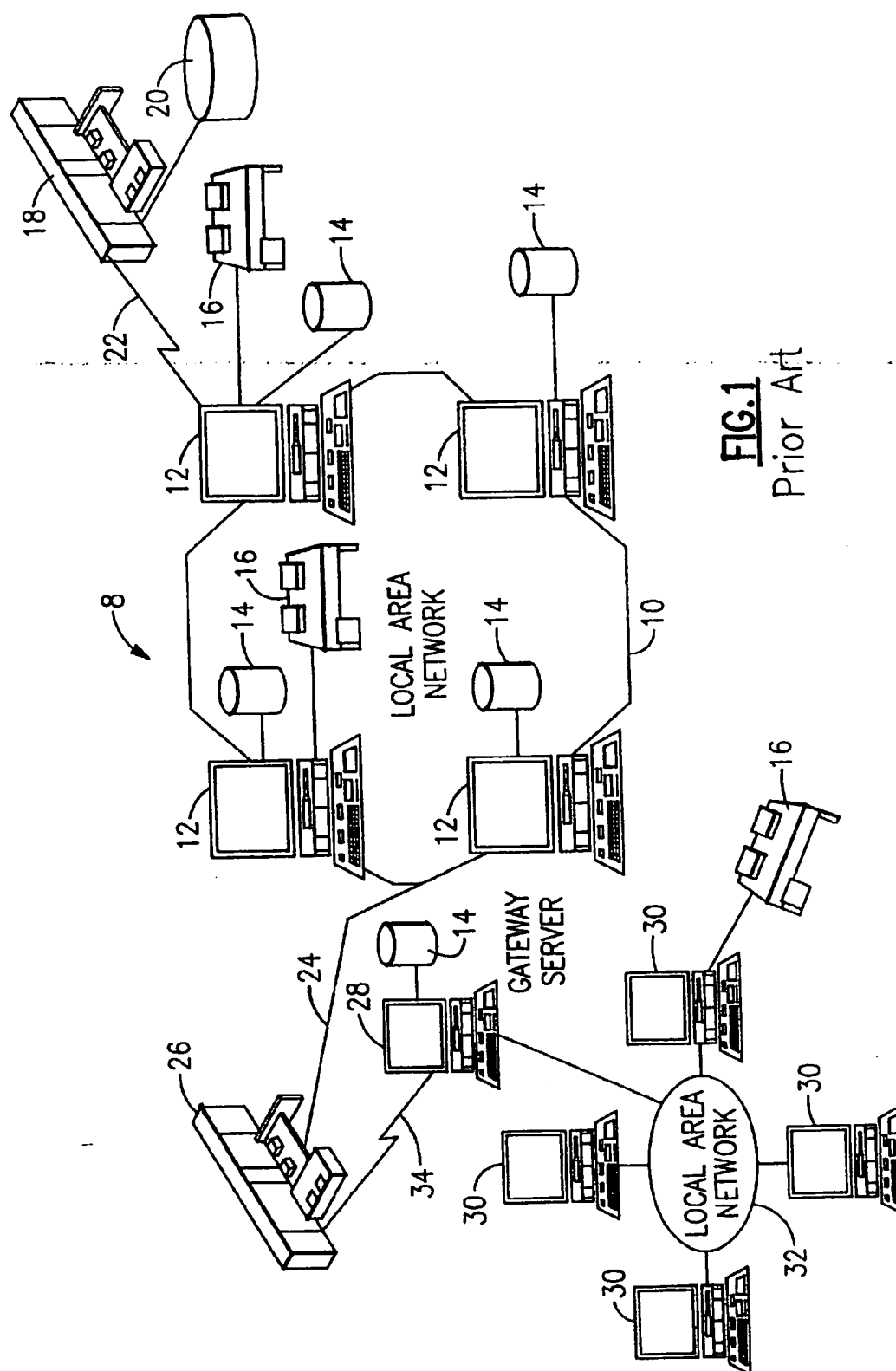


FIG. 1
Prior Art

FILE RETURN FILTER OWNER STATUS		HELP	
235	NAME		
236	LIBRARY		▽
237	TYPE		▽
238	VERSION		▽
239	LEVEL		▽

MODEL STATUS: VALID		MODEL OWNER: FORD	
240	NAME	TYPE	LIB
0001	A ALU	SCHEM	CP_LIB
0002	I ALU	VHDL	CP_LIB
0003	O ALU	SYNTHES	CP_LIB
0004	I ADDER	LAYOUT	TECH
0005	A MULT_DIV	LAYOUT	TECH
0006	I MULT1	GATE	TECH
0007	I DIV1	GATE	TECH
0008			
0009			
		VERS	LEVEL
		BASE	ENGIN
		BASE	ENGIN
		BASE	ENGIN
		HIPWR	QUAL
		LOPWR	QUAL
		LOPWR	QUAL
		LOPWR	QUAL
			STATUS
			VALID
			VALID
			VALID
			INVALID
			VALID
			VALID
			VALID

FIG.2

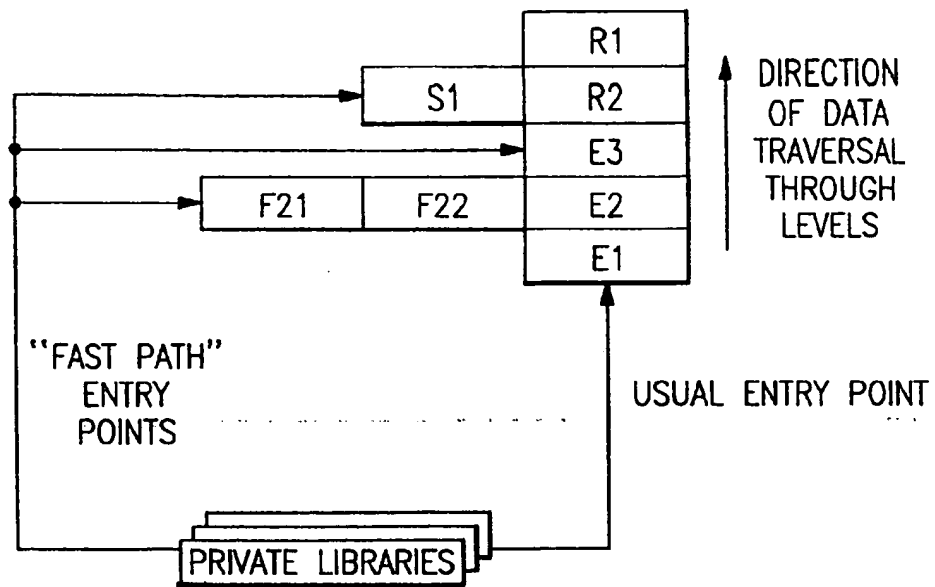


FIG. 3

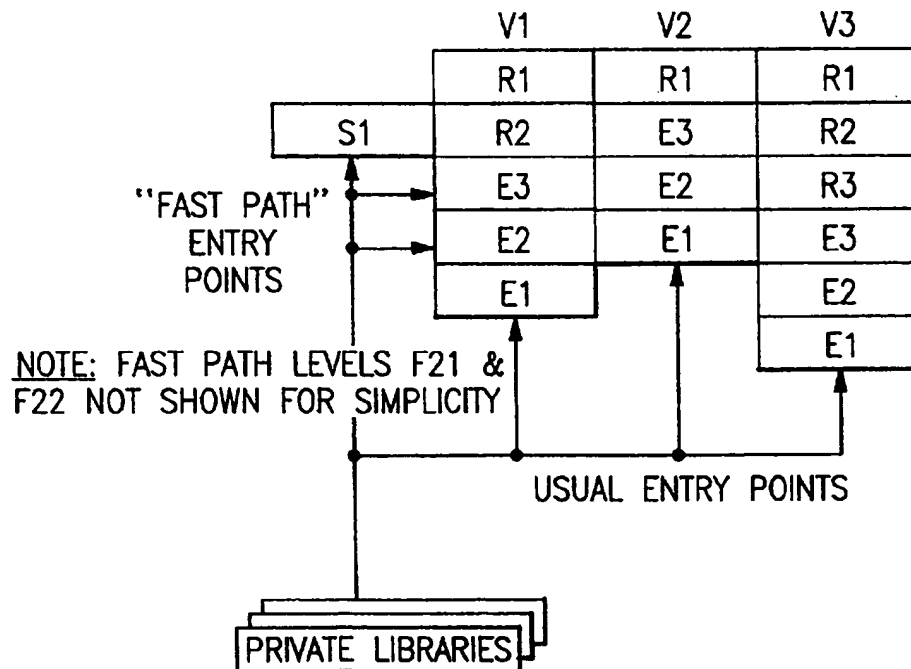
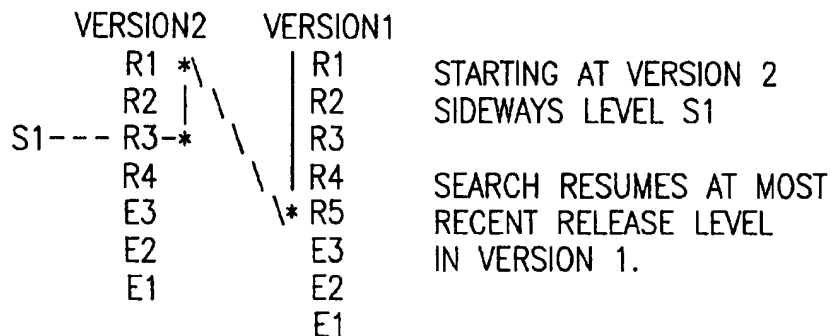
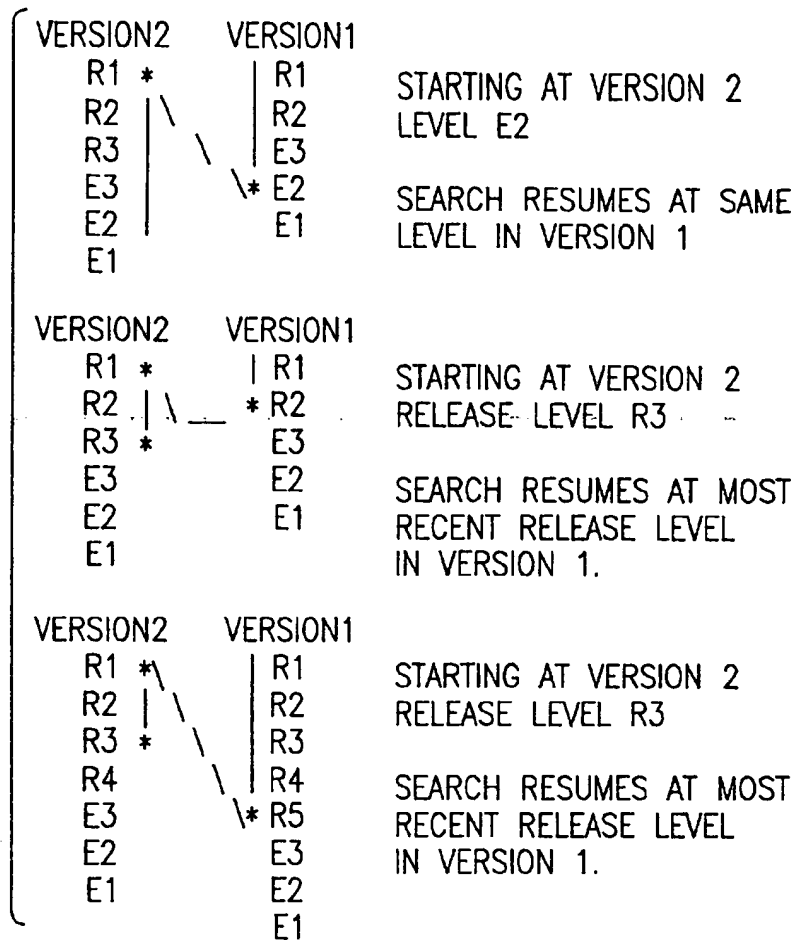
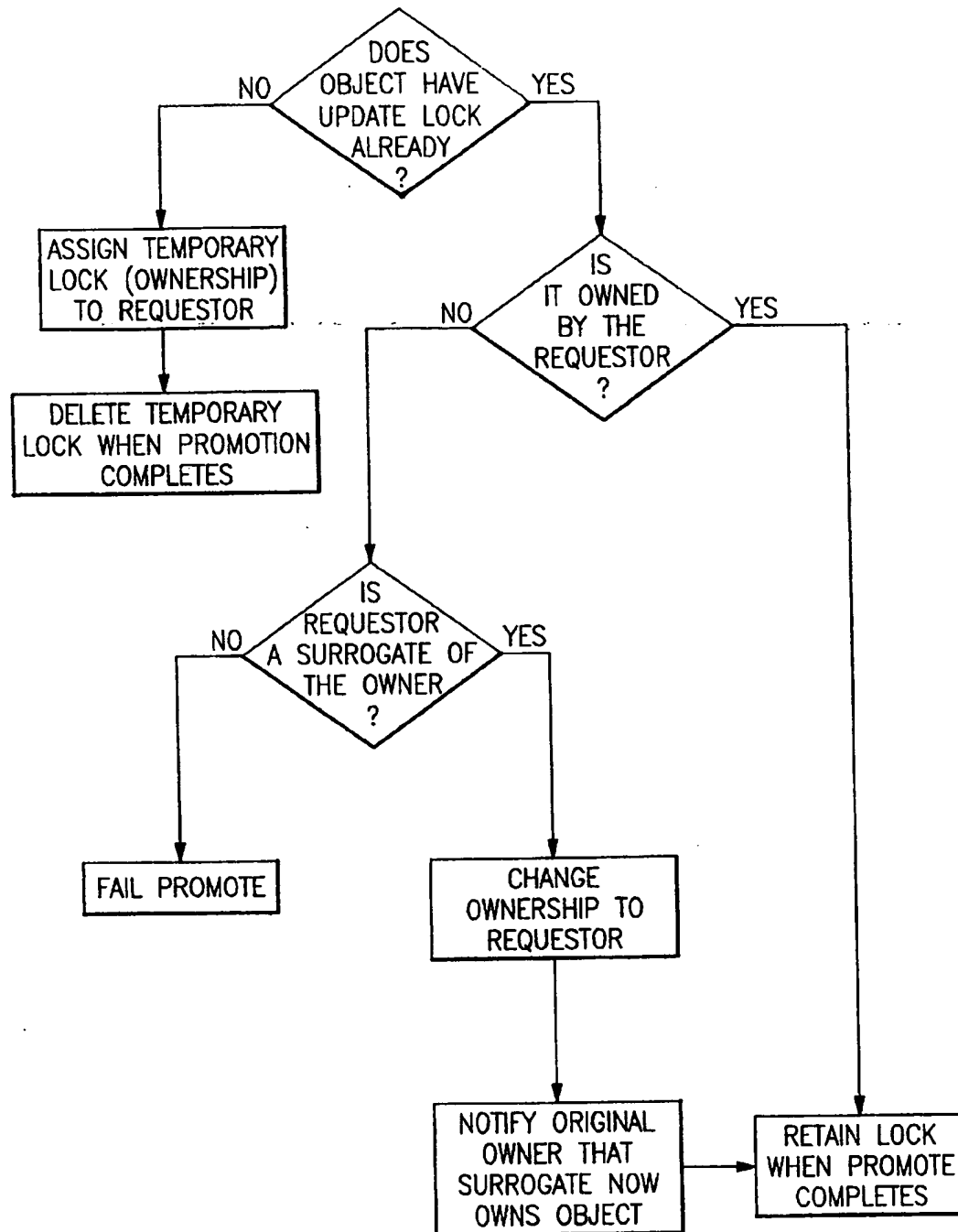
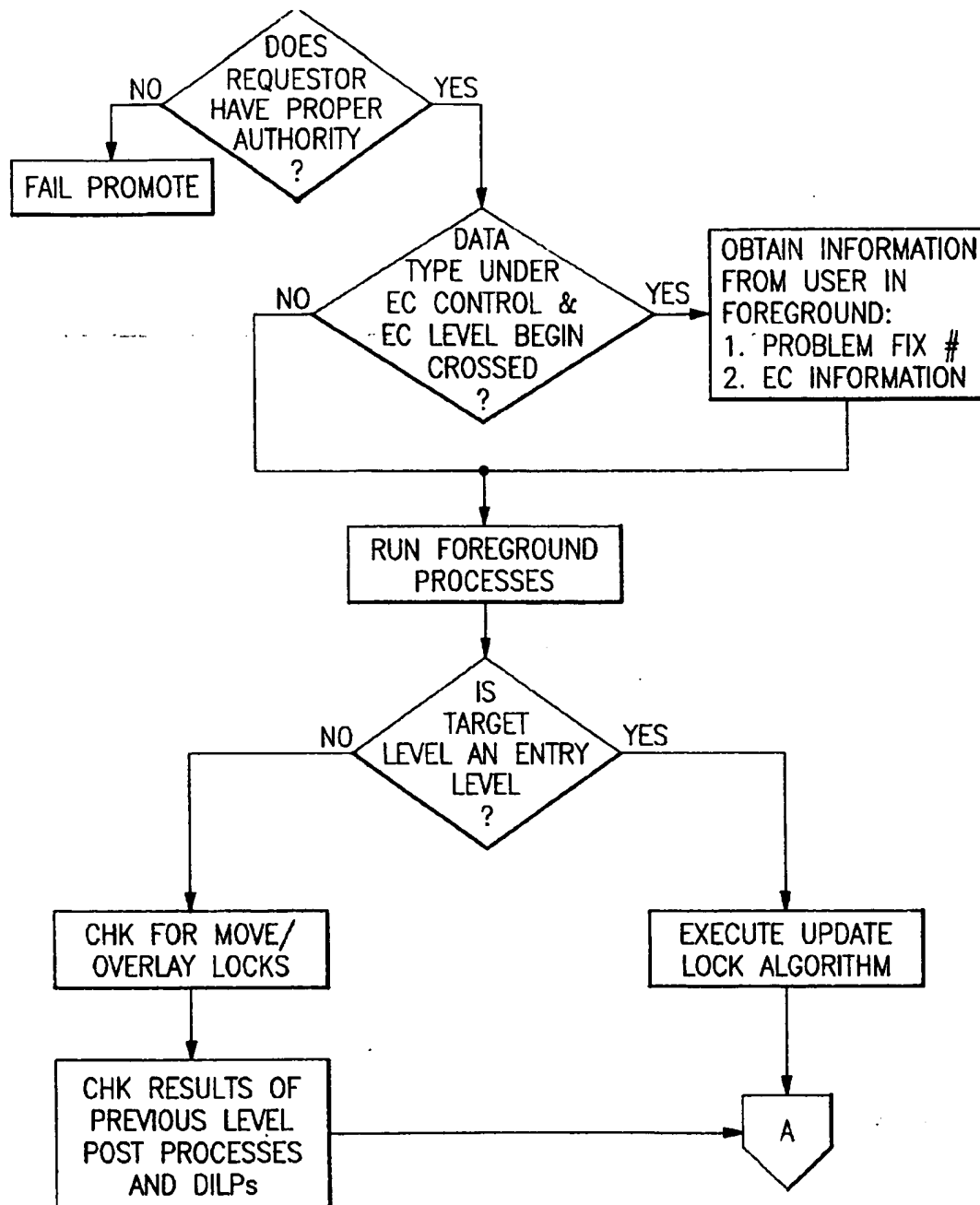
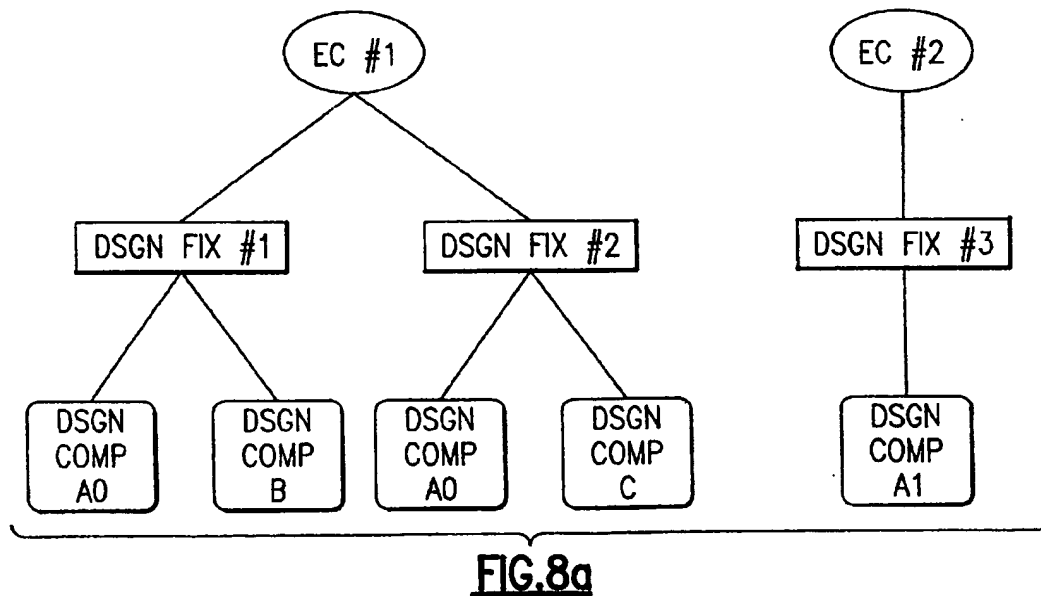
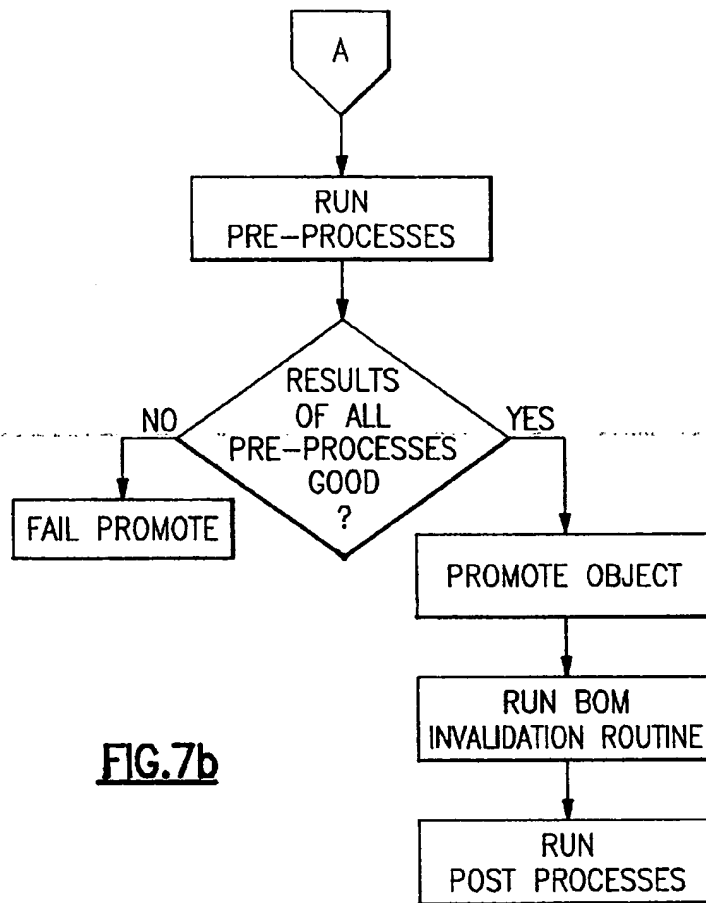


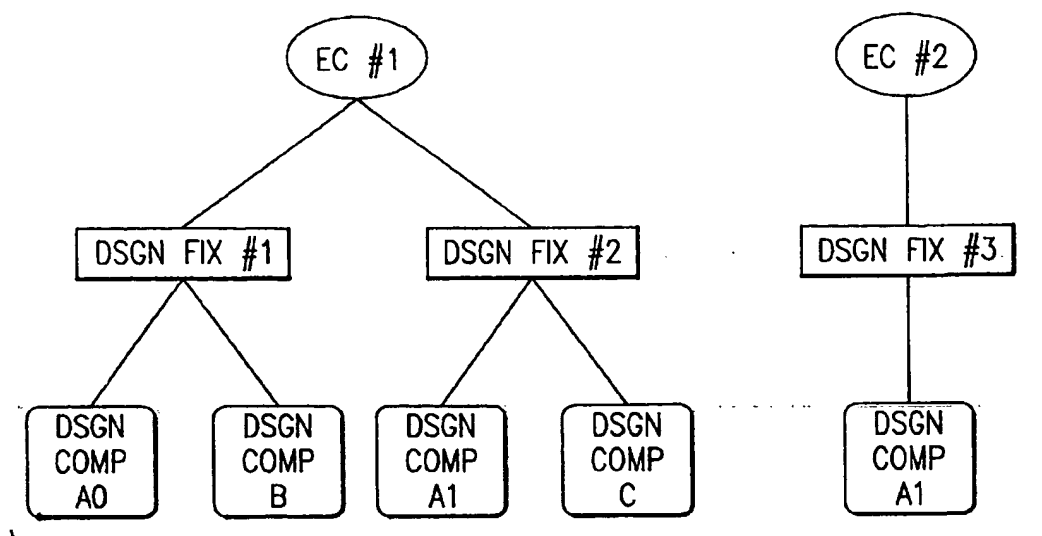
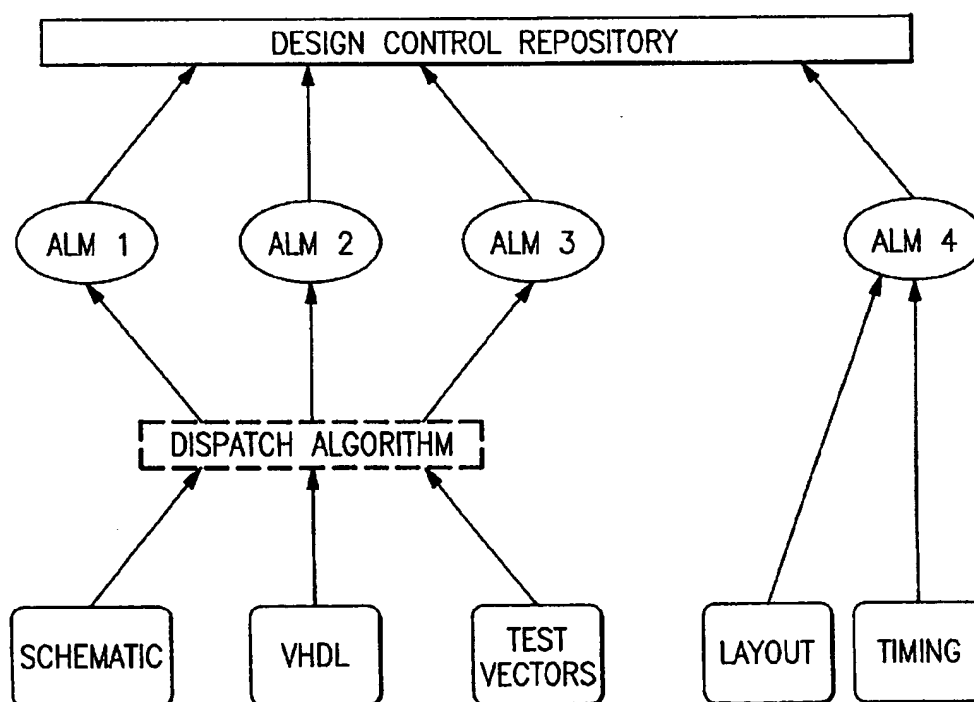
FIG. 4

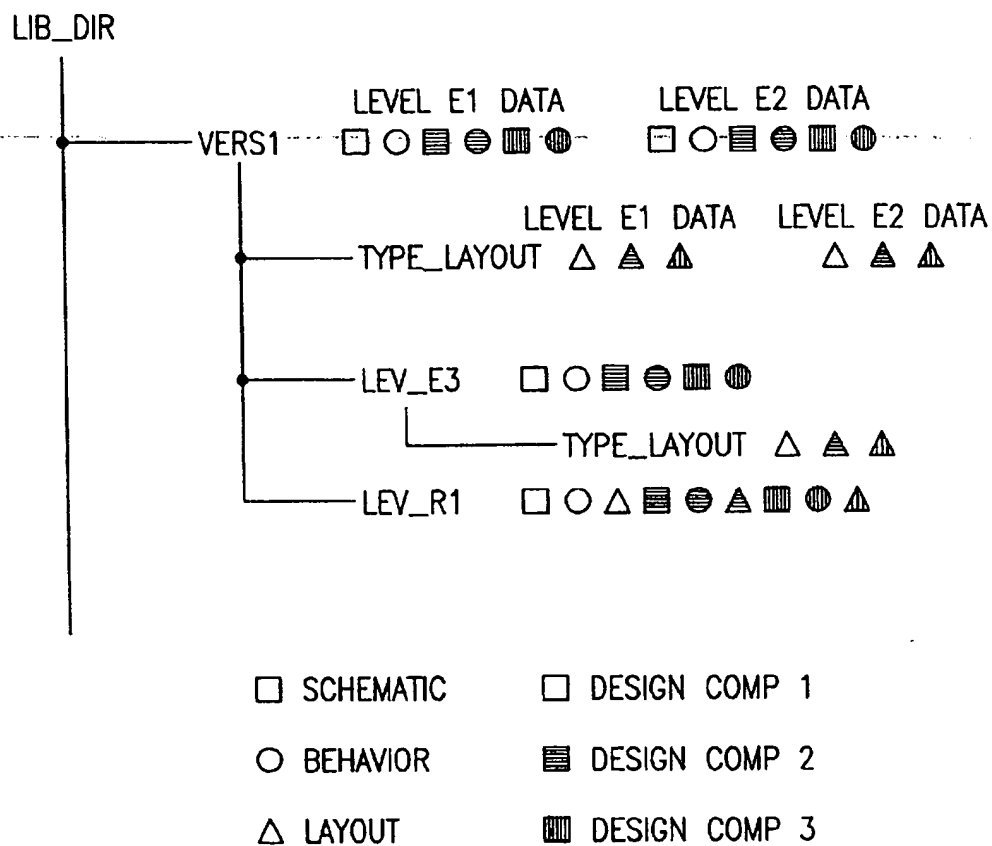
FIG. 5a**FIG. 5b**

**FIG. 6**

**FIG. 7a**

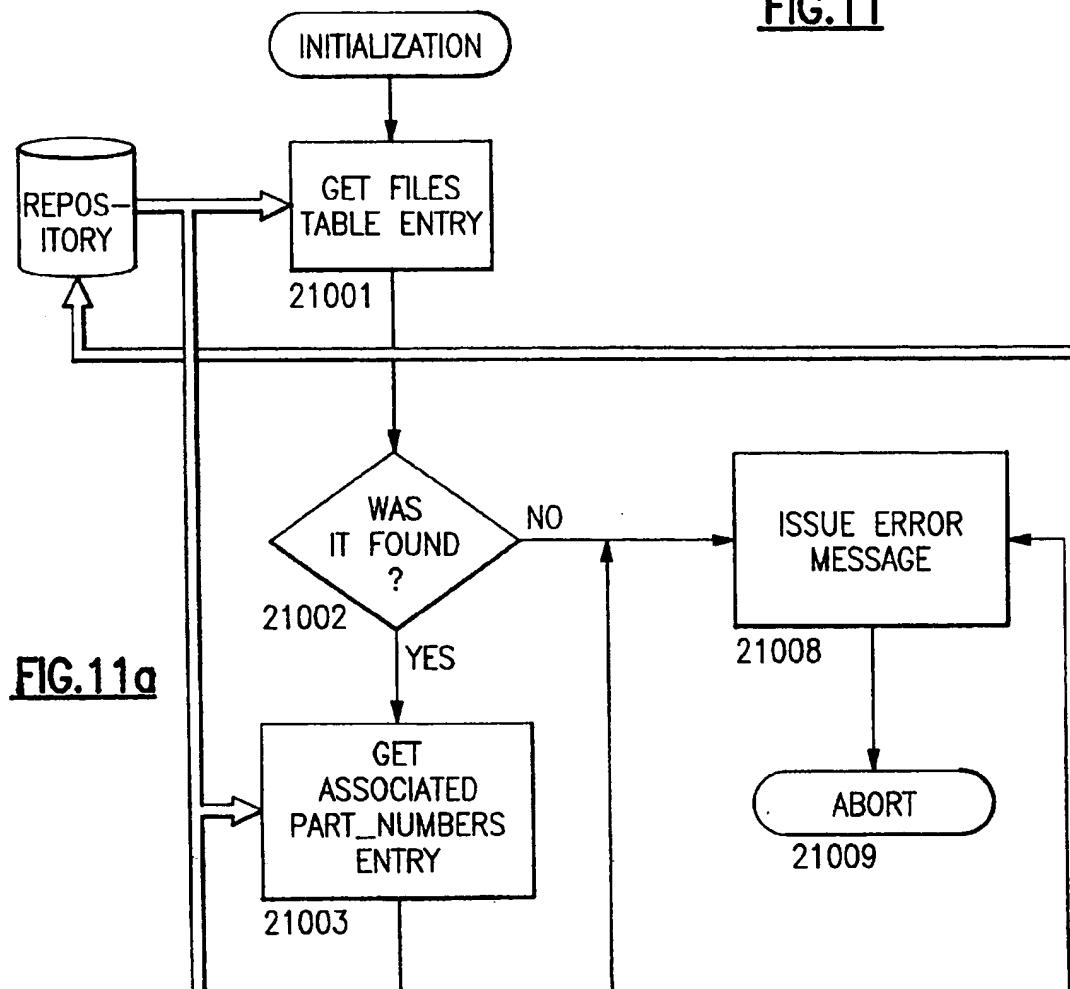


**FIG.8b****FIG.9**

**FIG.10**

11a	11c
11b	11d

FIG.11



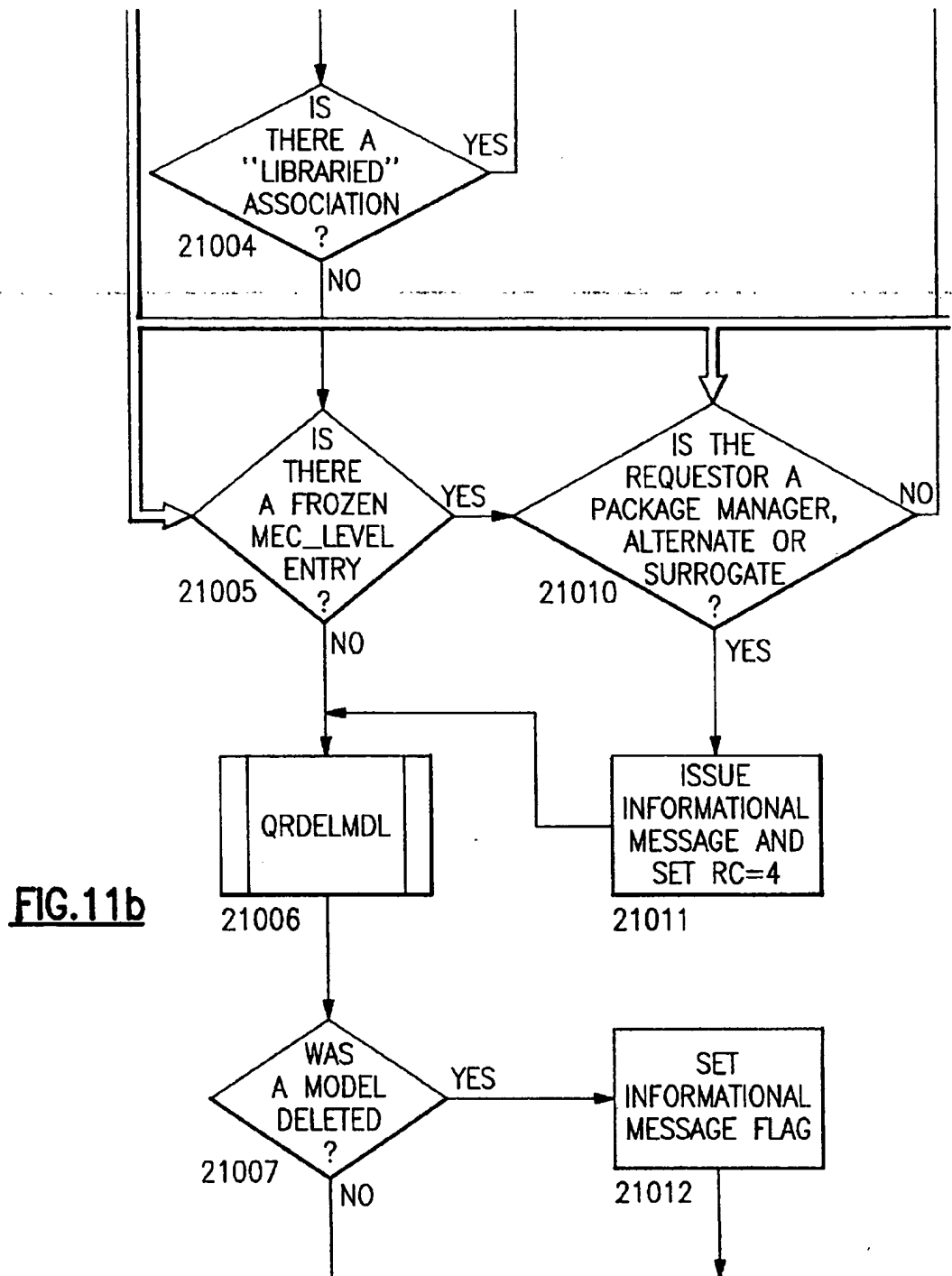
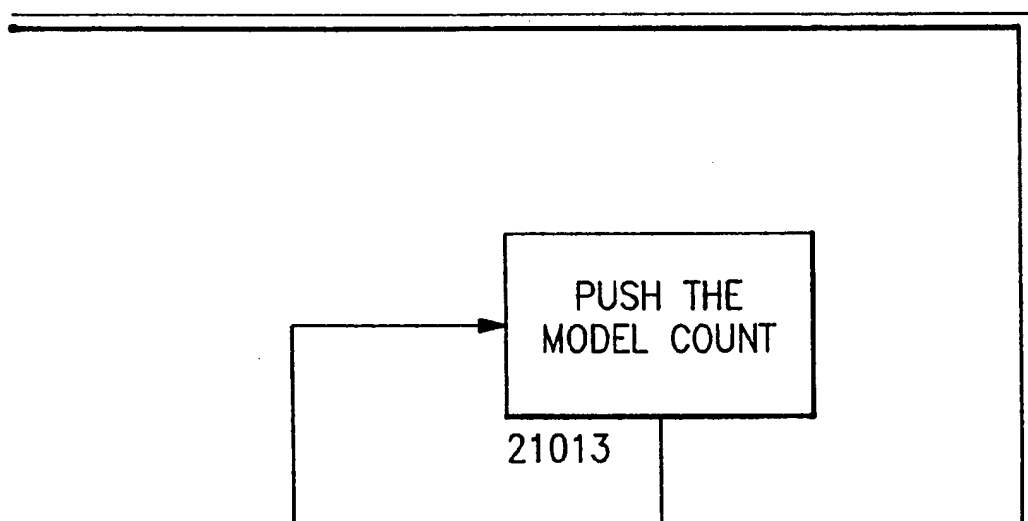


FIG. 11c



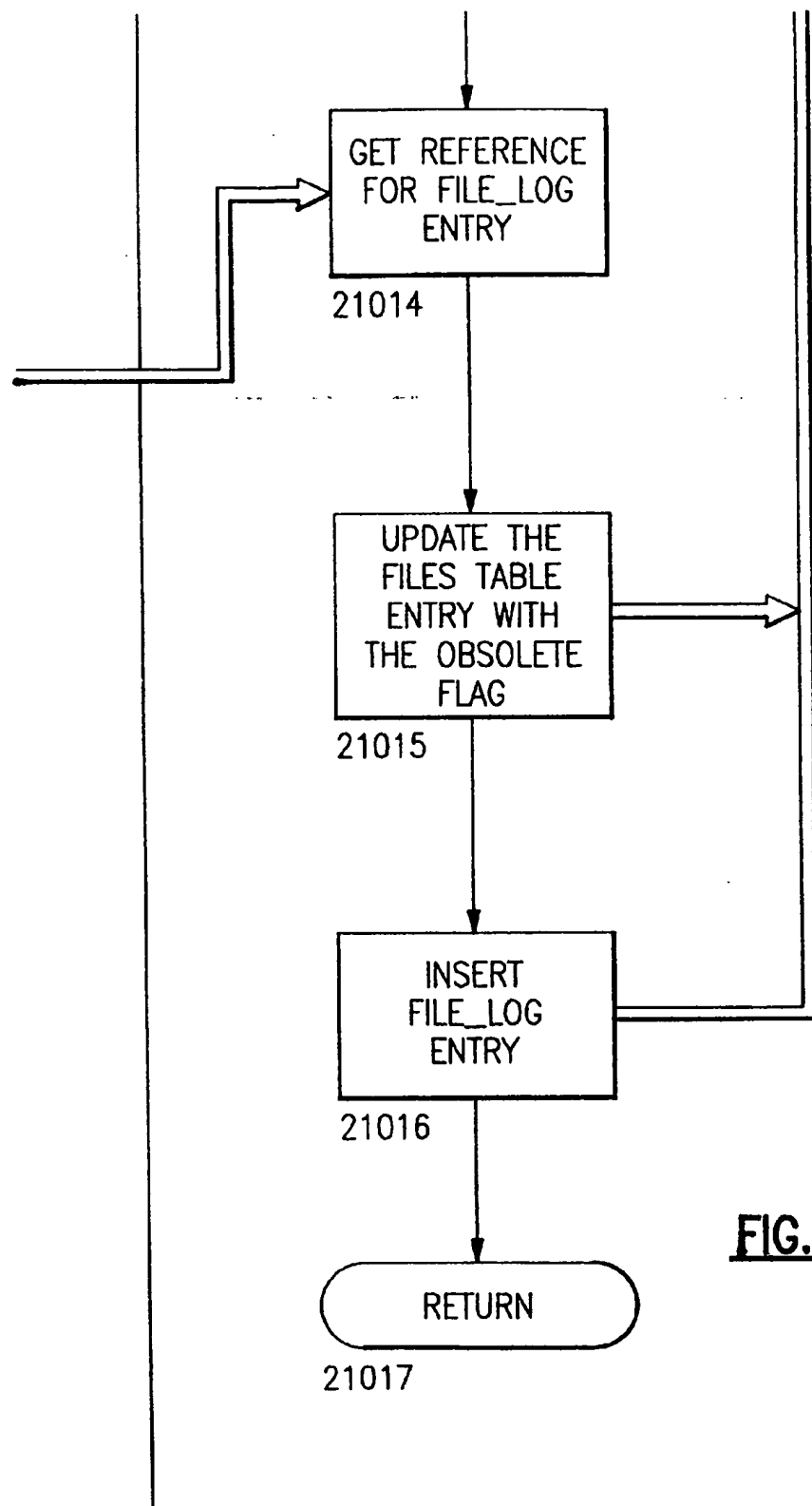
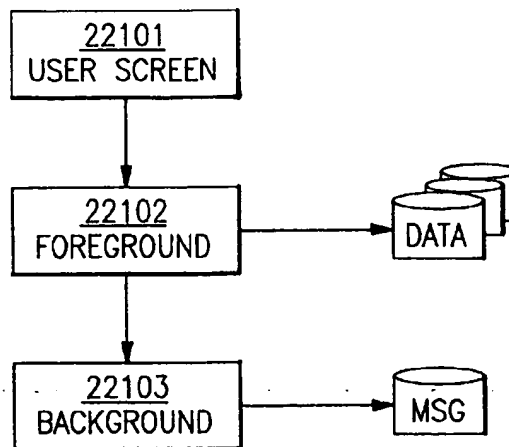
**FIG. 11d**

FIG.12

INITIATE LIBRARY PROMOTE

22201 NAME

22202 VERSION

22203 TYPE

22204 LIBRARY

22205 LEVEL

DEST ENTRY LEVEL

22206

22207

22208

22209 ☐ VIA LIST ☐ RESET UPDT LOCK

☒ FOREGROUND CHECKING ☐ HIGH PRIORITY

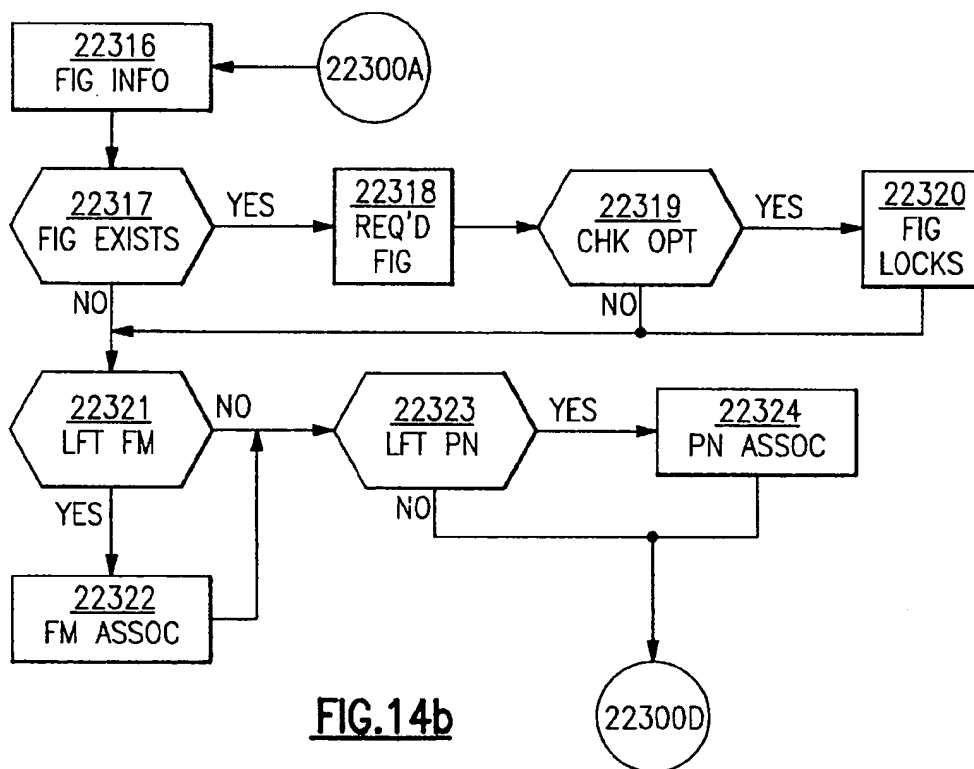
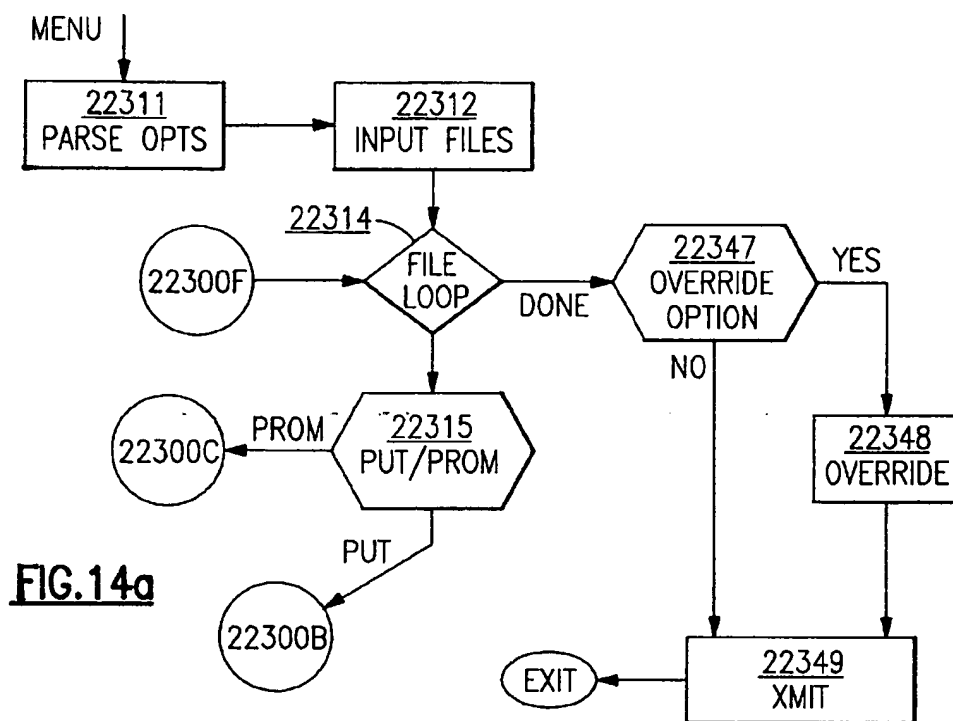
☒ VIA COPY ☐ OVERRIDE PARMS

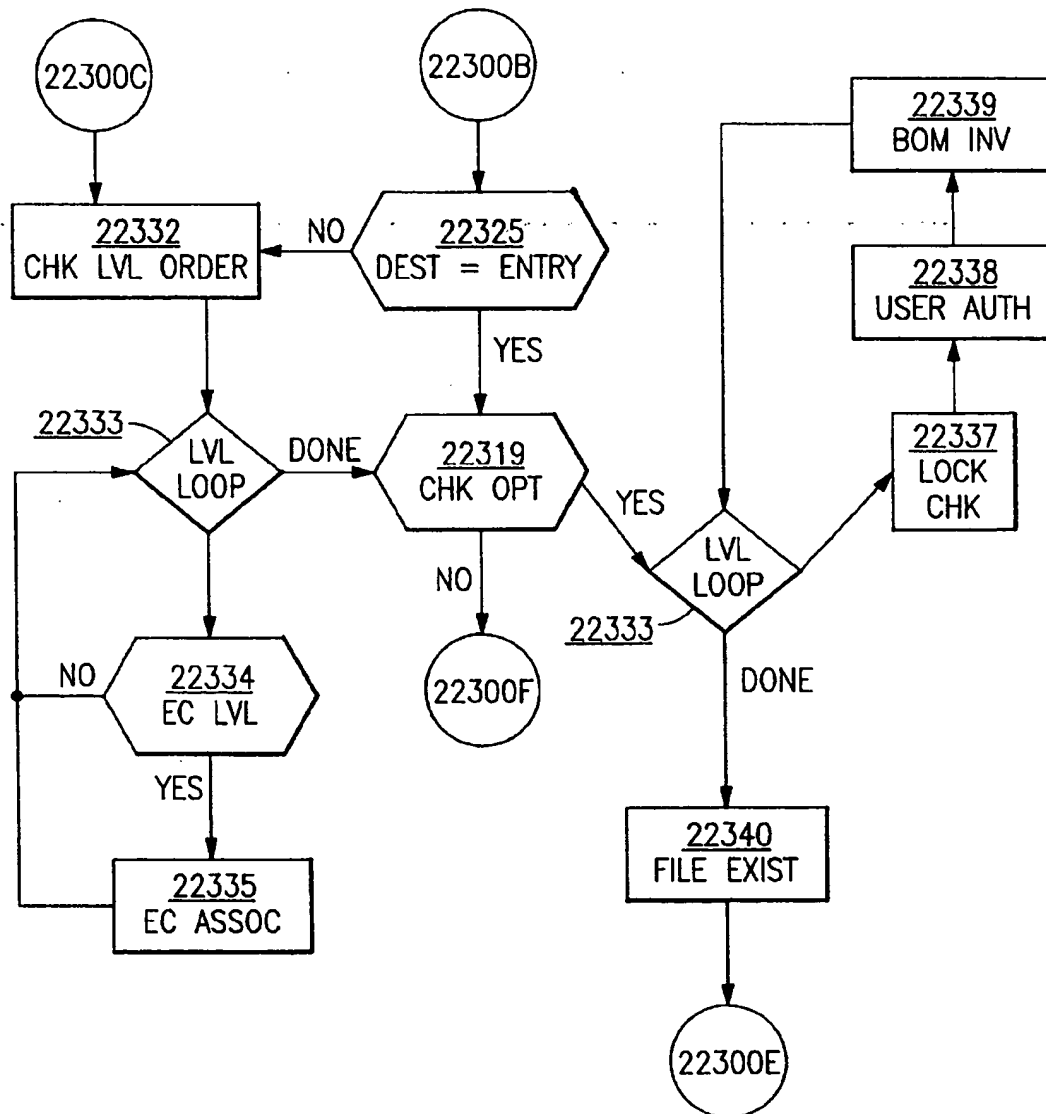
AVAILABLE TO CONTROLLED FILES ONLY

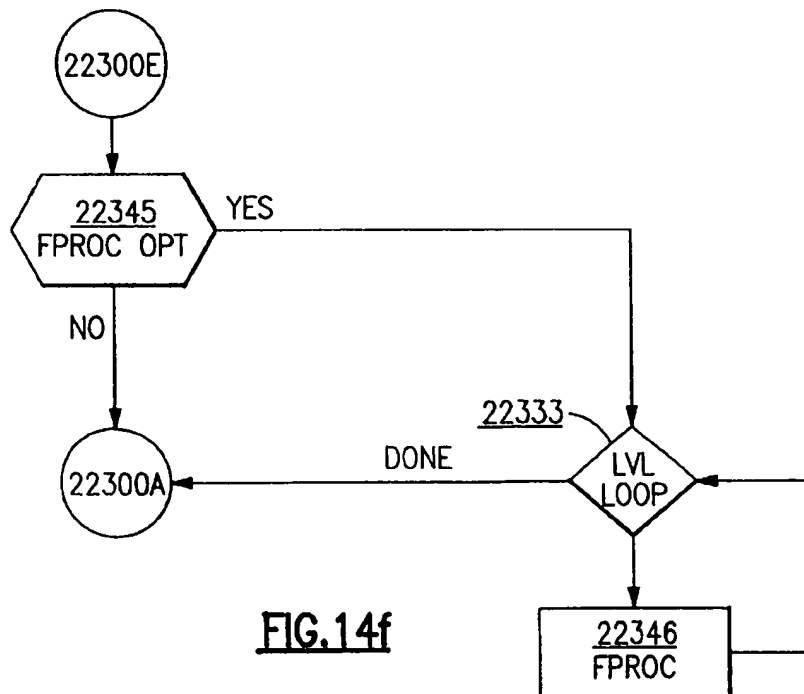
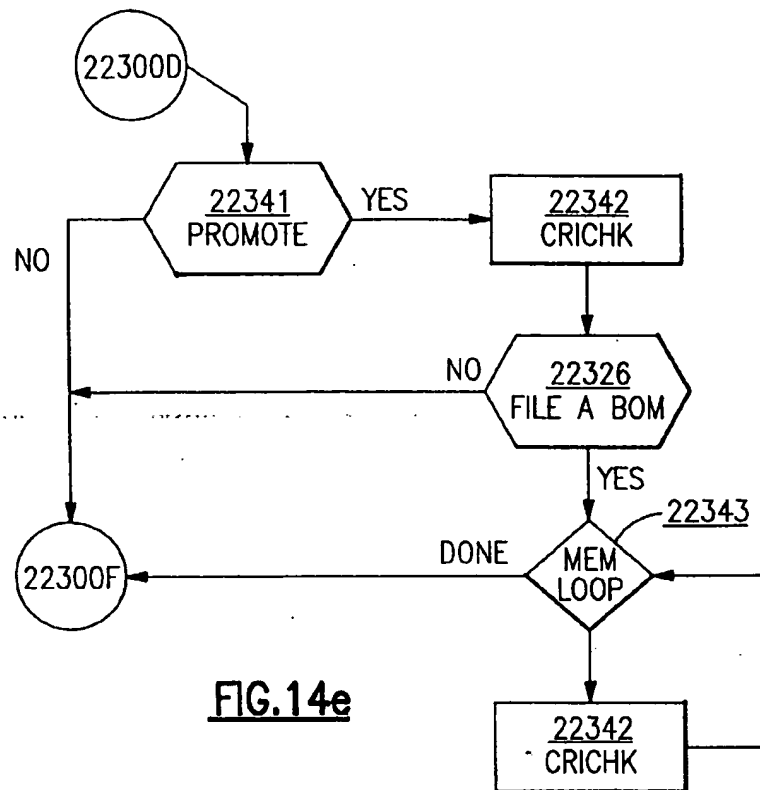
22210 ☐ BOM PROMOTE ☐ RETAIN SOURCE

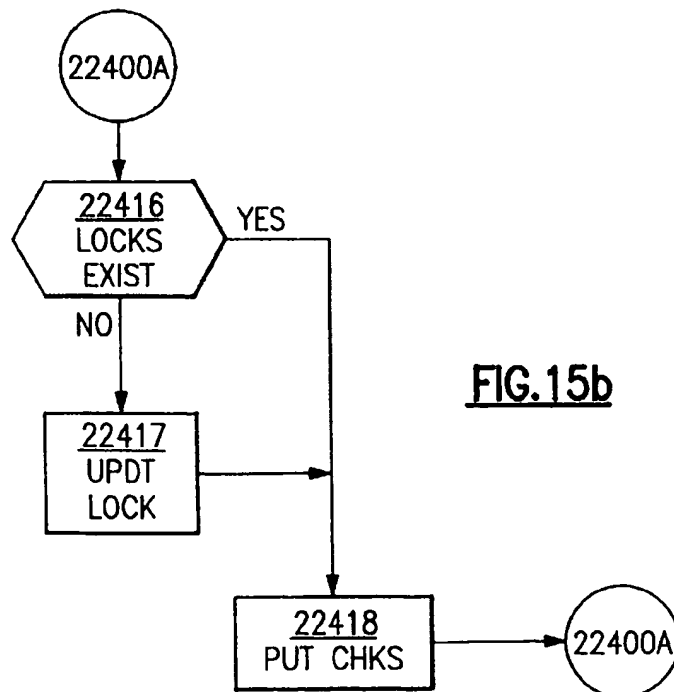
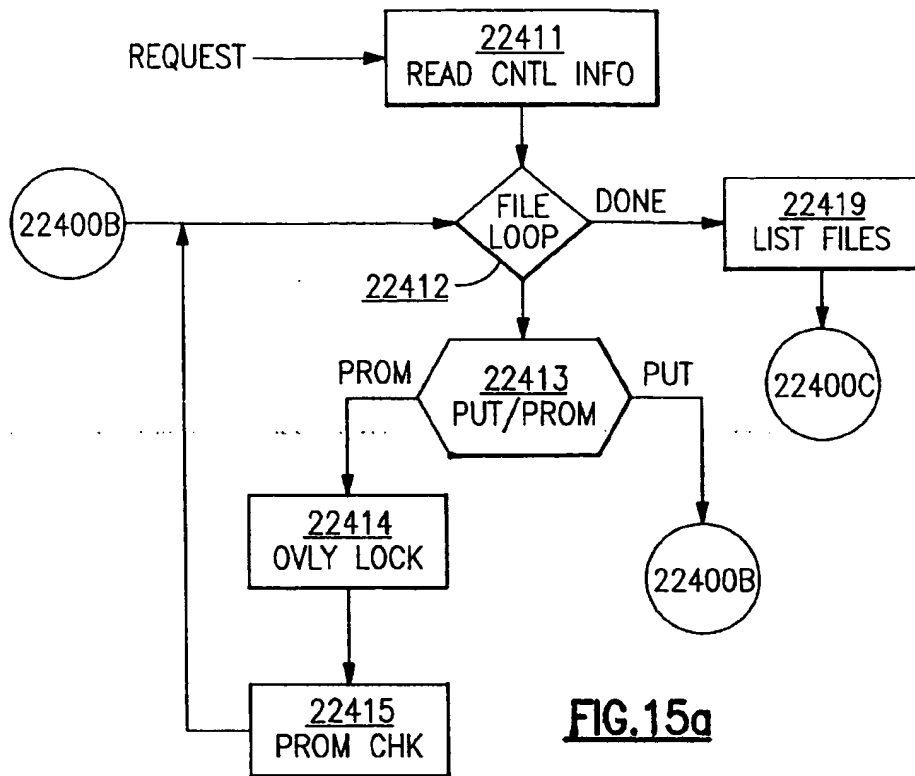
☐ ANCHOR ONLY

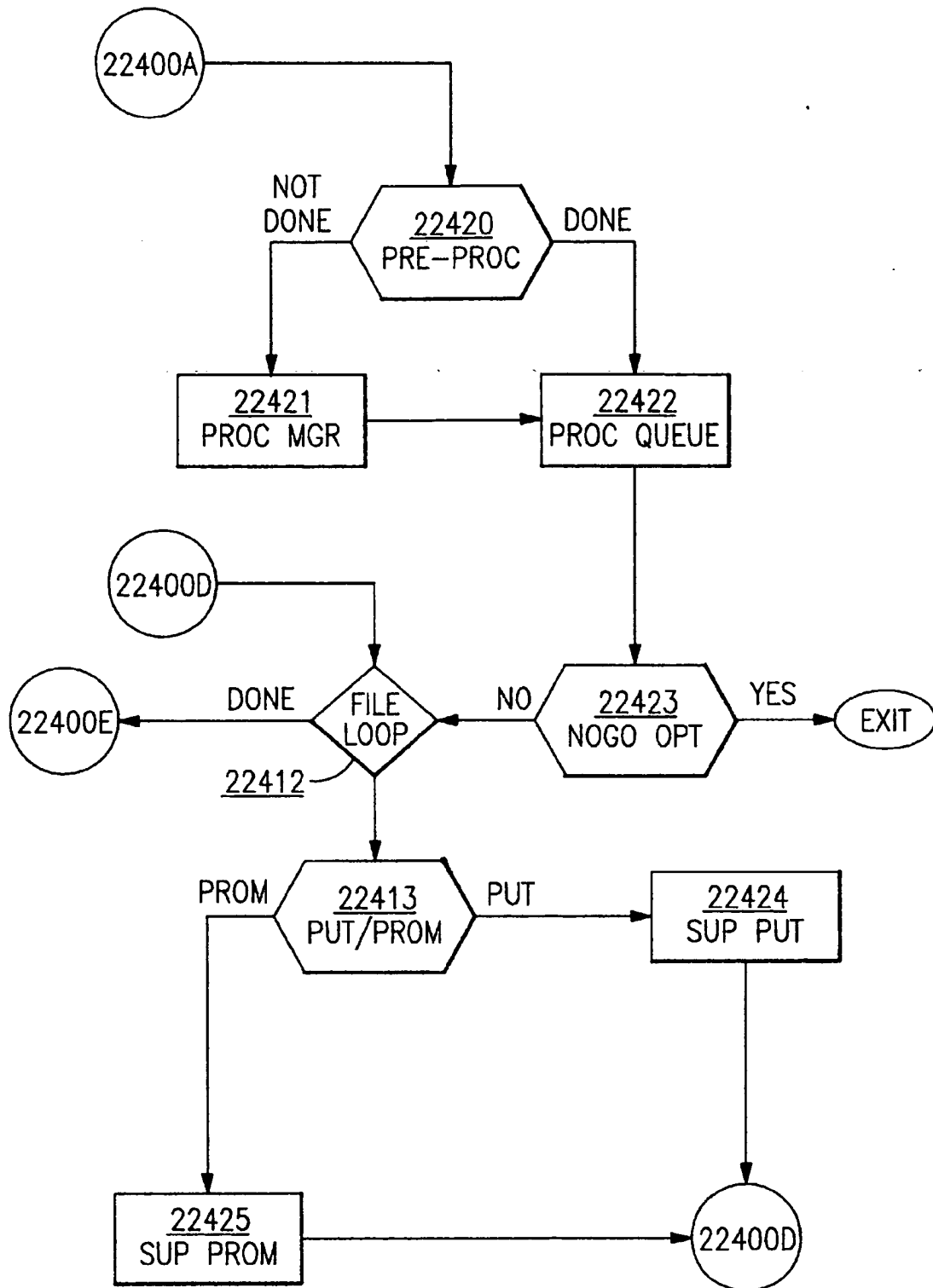
FIG.13

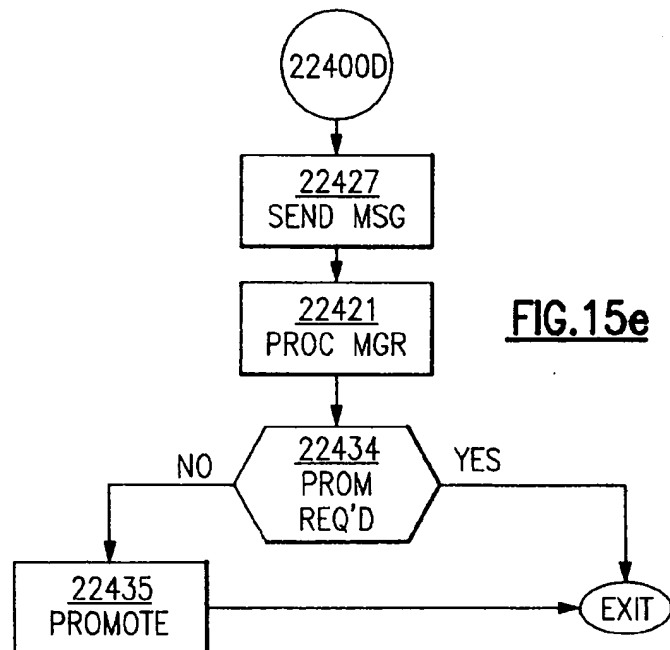
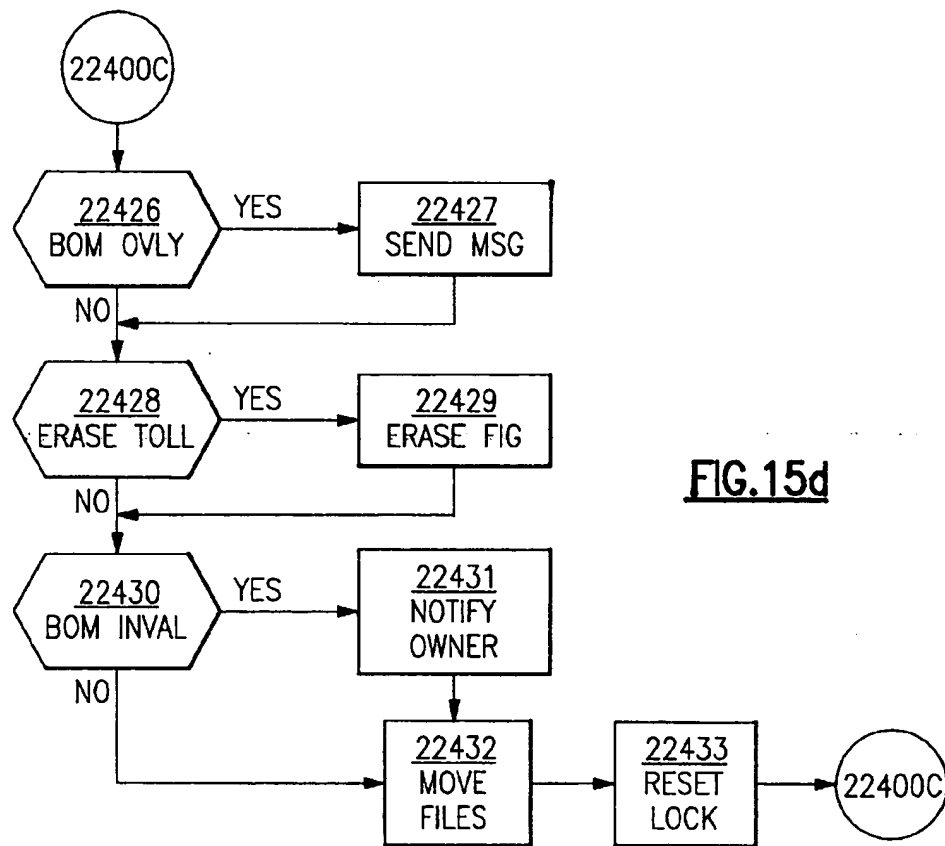


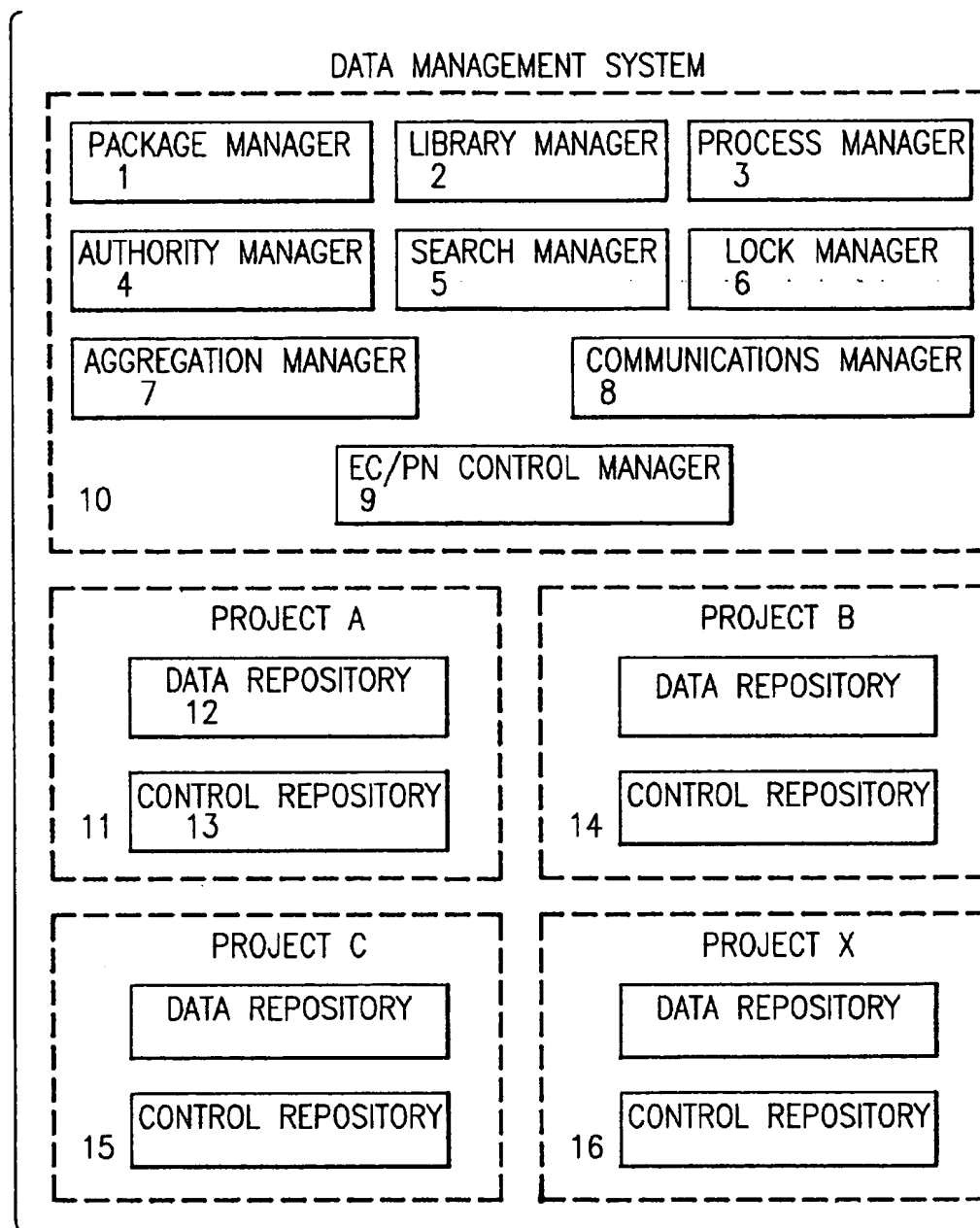
**FIG. 14d**





**FIG. 15c**



**FIG.16**

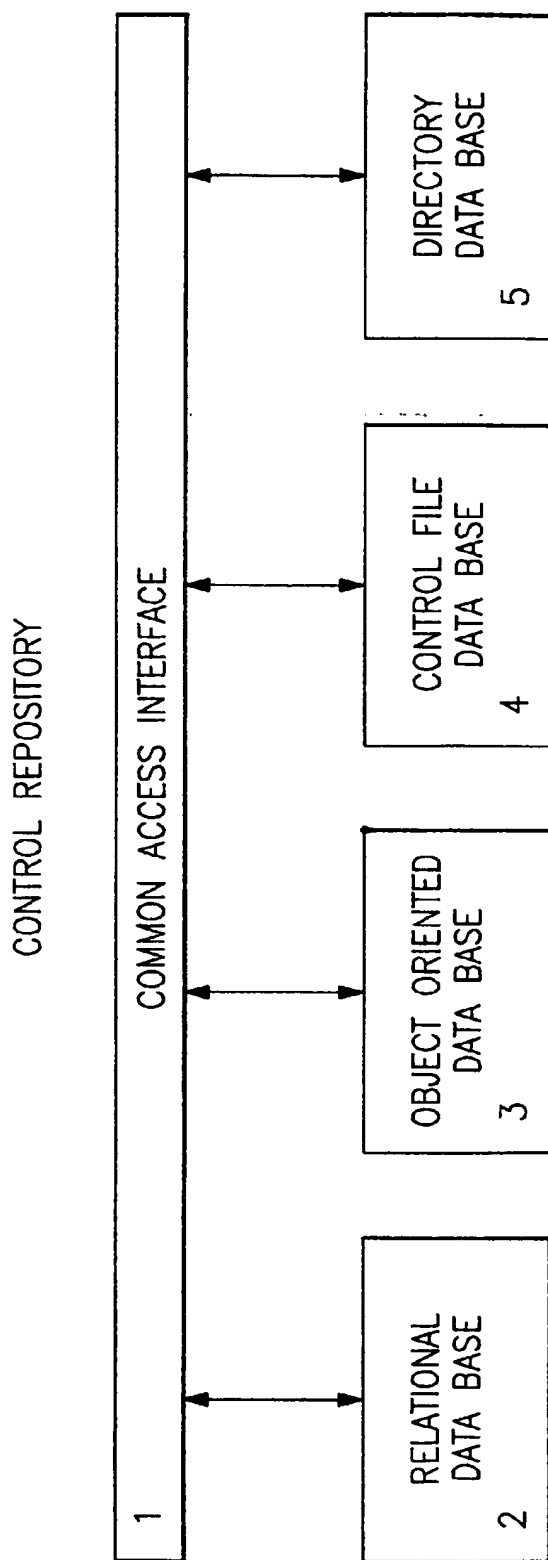


FIG.17

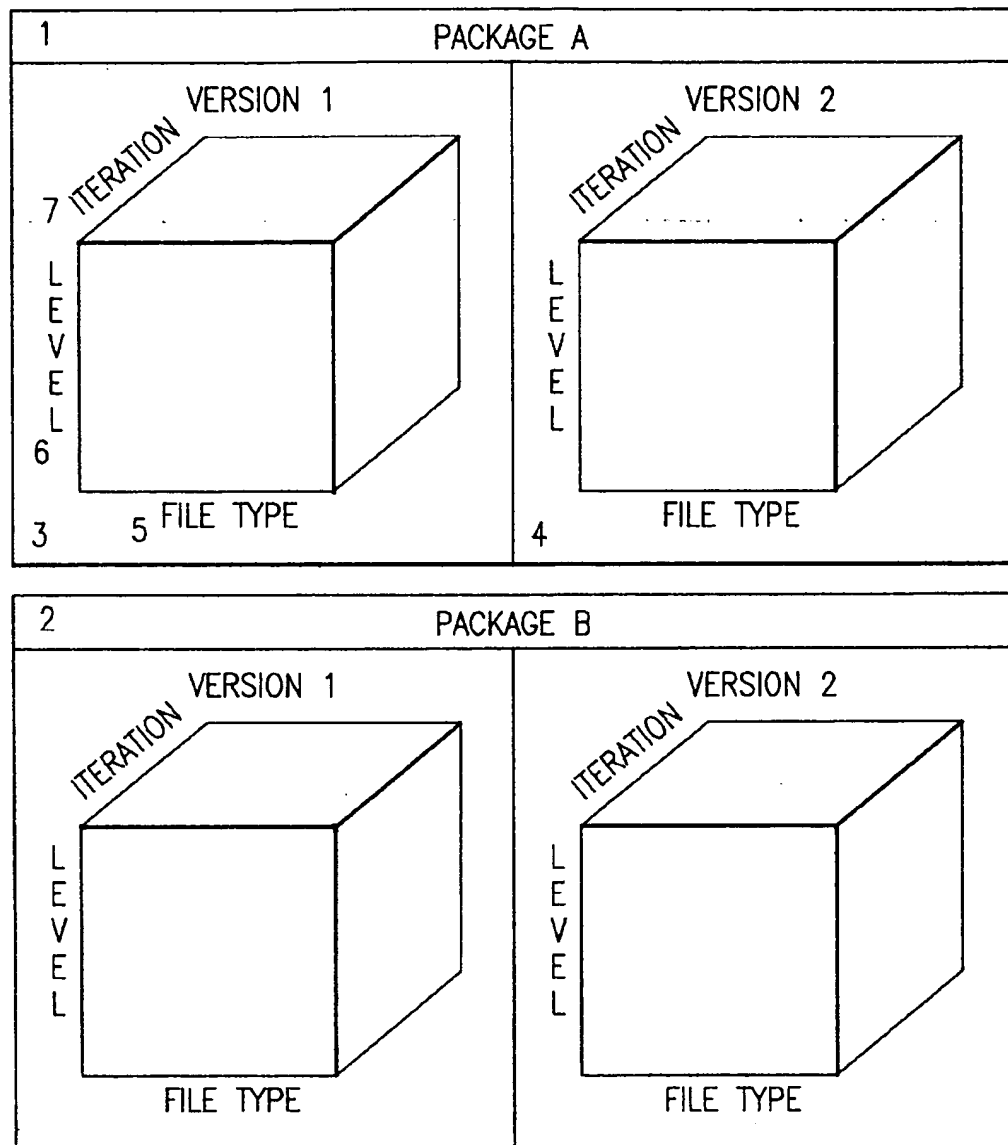
**FIG.18**

Figure 19

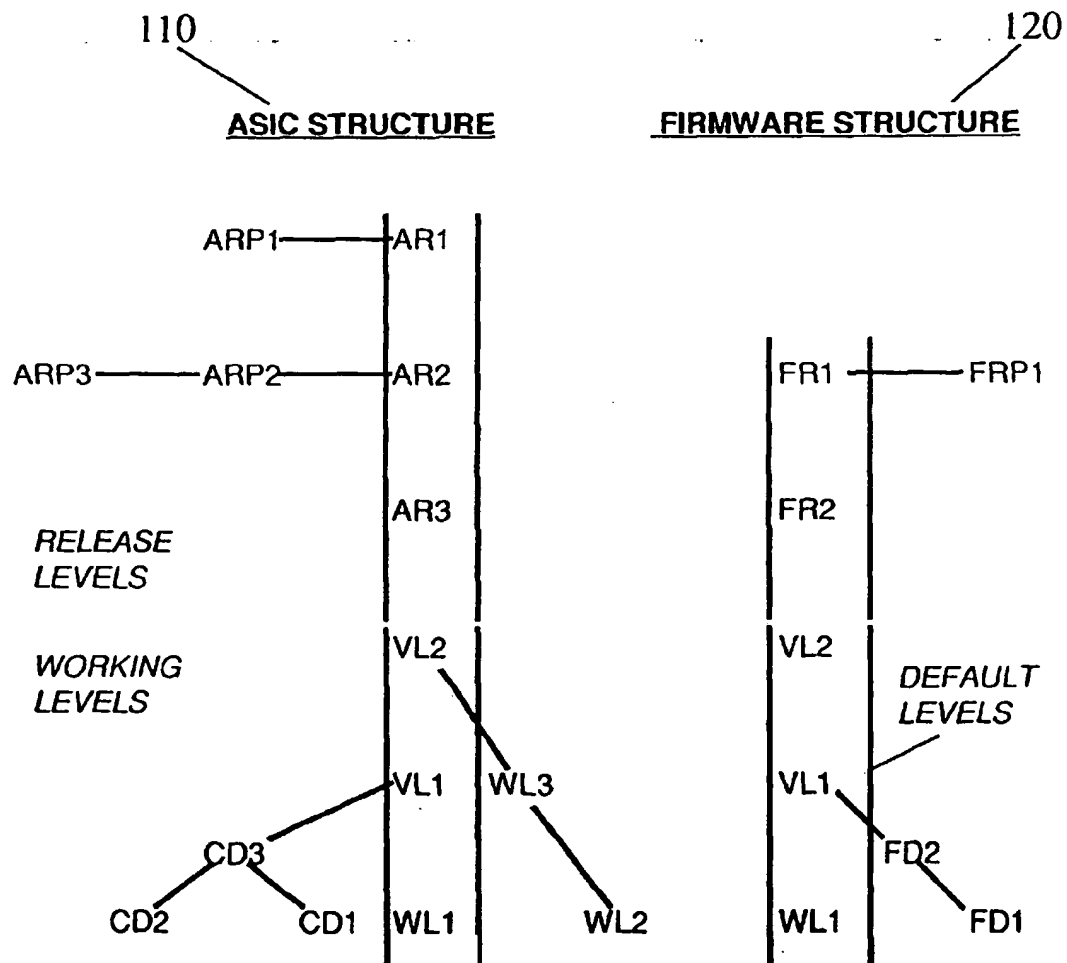


Figure 20

152	XXX/XX/00	WL1	NONE	NONE
	XXX/XX/WL1	VL1	YY	REPOS1...
	XXX/XX/VL1	VL2	NY	REPOS1
153	ASIC/XX/CD1	CD3	YY	REPOS1
	ASIC/XX/CD2	CD3	YY	REPOS1
	ASIC/XX/CD3	VL1	NY	REPOS1
	ASIC/XX/WL2	WL3	YY	REPOS1
	ASIC/XX/WL3	VL2	NY	REPOS1
	ASIC/V1/VL2	AR3	NY	REPOS2
154	ASIC/V1/AR3	AR2	NN	REPOS2
	ASIC/V1/AR2	AR1	NN	REPOS2
	ASIC/V1/AR1	***	NN	REPOS2
	ASIC/V1/ARP3	ARP2	YN	REPOS2
	ASIC/V1/ARP2	AR2	YN	REPOS2
	ASIC/V1/ARP1	AR1	YN	REPOS2
	FIRM/XX/FD1	FD2	YY	REPOS1
155	FIRM/XX/FD2	VL1	NY	REPOS1
	FIRM/V1/VL2	FR2	NY	REPOS3
156	FIRM/V1/FR2	FR1	NN	REPOS3
	FIRM/V1/FRP1	FR1	YN	REPOS3
	FIRM/V1/FR1	***	NN	REPOS3

COMPUTERIZED DESIGN AUTOMATION METHOD USING A SINGLE LOGICAL PFVL PARADIGM

COPYRIGHT NOTICE AND AUTHORIZATION

This patent document contains material which is subject to copyright protection.

(C) Copyright International Business Machines Corporation 1995, 1996 (Unpublished). All rights reserved. Note to US Government Users—Documentation related to restricted rights—Use, duplication, or disclosure is subject to restrictions set forth in any applicable GSA ADP Schedule Contract with International Business Machines Corporation.

The owner, International Business Machines Corporation, has no objection to the facsimile reproduction by any one of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records of any country, but otherwise reserves all rights whatsoever.

FIELD OF THE INVENTION

This invention is related to a Data Management Control System and Methods for Computerized Design Automation (CDA) Applications, and particularly to methods useful for concurrent engineering in connection with the design, development and manufacturing of complex electronic machines such as computer systems and their complex electronic parts.

GLOSSARY OF TERMS

While dictionary meanings are also implied by certain terms used here, the following glossary of some terms may be useful.

AFS Andrew File System—File Management System developed by Transarc Inc. and used on Unix/AIX Networks.

API Application Program(ming) Interface.

ASC Accredited Standards Committee (ANSI)

BOM Bill of Materials

CIM Computer Integrated Manufacturing

CR Control Repository

CRC Cyclic Redundancy Check

CSI Compiler VHDL Analyzer developed by Compass Design Systems

DCS Design Control System. Our Design Control System incorporates Data Management System Processes, including interactive data management systems which supply processes which may be applicable in general data management systems, such as a process manager, a promotion manager, a lock manager, a release manager, and aggregation manager and the other processes we describe herein as part of a Computer Integrated Design Control System and, where the context applies, Data Management System, is a Data Management System functioning within an overall integrated design control system.

DILP Designer Initiated Library Process

DM Data Manager or Data Management

DMCU Data Management Control Utilities

DMS Data Management System

DR Data Repository

EC Engineering Change

EDA Electronic Design Automation

GUI Graphical User Interface

PDM Product Data Management

PIM Product Information Management

PN Part Number

RAS Random Access Storage

sim static inline memory

tape-out Delivery of a coherent set of design data to manufacturing. Also known as Release Internal Tape (RIT) within IBM.

TDM the Cadence Team Design Manager (most currently Version 4.4)

VHDL Very High-level Design Language—A high level language comprised of standards supported by IEEE and the EDA industry. The language is widely used in the electronics and computer industry and by the military as an alternative to Verilog and ADA, other high level computer coding languages.

BACKGROUND OF THE INVENTION

As Data Management systems grow more complex, they have more users interacting with them, and issues such as performance, data integrity, workload management, batch processing, efficiency and continuous availability need to be solved. Many systems on the market today can only handle small numbers of users simultaneously, offer little or no expansion capabilities and frequently require manual intervention to process data through the system. In addition, the mechanisms for maintaining data integrity, ownership, and file management are either limited in capability or are unable to prevent loss of data and/or collisions under certain conditions. With the growing presence of distributed computing, and the increased need for sharing large amounts of data across an enterprise, a solution is required to address these problems for a computer integrated design control system for concurrent engineering and other applications.

In the article entitled "Beyond EDA (electronic design automation)", an example of one kind of computerized design automation (CDA), published in Electronic Business Vol.19, No.6 June 1993 P42-46, 48, it was noted that while billions of dollars have been spent over the past (then and still last) five years for electronic design automation systems (EDA) and software to help companies cut their design cycle, a huge gulf remains between design and manufacturing. To eliminate the gulf and thus truly comply with the commandments, companies are extending the concept of concurrent engineering to enterprise wide computing. The concept, which calls for integrating all the disciplines from design to manufacturing is becoming the business model of the 1990s. Achieving an enterprise wide vision requires tying together existing systems and programs and managing the data that flows among them. Software that makes that linkage possible is largely in the class known by two names: product data management (PDM) or product information management (PIM). Mr. Robinson, the author, described the experiences of several companies with PIM and PDM, in particular Sherpa and Cadence.

The design of complex parts, such as integrated circuits, computers, or other complex machines in a complete manufacturing operation like IBM's requires computer capability, with computers capable of processing multiple tasks, and allowing concurrent data access by multiple users. The IBM System 390 operating system known as Multiple Virtual Storage (MVS) allows such things as relational database management methods, such as the TIME system described by U.S. Pat. No. 5,333,316, to be used to reduce design time.

The TIME system is used within IBM for the purposes described in the patent during circuit design. However, these prior efforts treated design as directed to an entity and did not achieve the efficiencies provided by the system detailed in our description of our invention, which also can run under MVS, but also under other operating systems. Our detailed description of our invention will illustrate that we have furthered the objects of the invention of U.S. Pat. No. 5,333,316 by increasing the flexibility of a number of circuit designers who may concurrently work on designing the same integrated circuit chip and reducing the interference between chip designers. With the prior system, a user (a person, processor or program capable of using data in a relational database) was given a private copy of the master table. Alteration of a row in the user table was not automatically updated in the master table, because a lock mechanism prevented the row update, but that was a great improvement at the time, because no longer did multiple users have to wait for copying of a table, each time data from a user needed to be updated. This row locking and treatment of data has become widespread in the relational database field, and it has been enabled for use with multiple instances of a platform even on Unix machines today. We should note that also in the MVS art, there have been proposed various library systems, e.g. those represented by U.S. Pat. Nos. 5,333,312 and 5,333,315 and others which relate to IBM's Image Object Distribution Manager in the ImagePlus product line of IBM, and IBM's Office Vision are examples of systems enabling control of a source document while allowing access by multiple users. Implementation of these patented ideas enable synchronous and asynchronous copying of a document into a folder in a target library. These methods provide for check out of a document and its placement in a target library while locking the document in the source library to prevent changes while the checked out document is out. But these steps are only some of the many things that are needed to bring a product to a release state. Bringing a product to a release state is an object of the current developments relating to design control in a manufacturing setting.

Concurrent engineering is required among many engineers working in parallel and at different locations worldwide. Furthermore, as noted by Oliver Tegel in "Integrating human knowledge into the product development process" as published in the Proceedings of the ASME Database Symposium, Engineering Data Management: Integrating the Engineering Enterprise ASME Database Symposium 1994. ASCE, New York, N.Y., USA. p 93-100, specialists who are not working directly together are often needed for solving the demanding tasks that arise during the development of today's advanced products. During product development, assistance is required from other departments such as manufacturing, operations scheduling, etc. Even the vendors and customers should be integrated into the product development process to guarantee the product developed will be accepted in the market.

There is a need for integrators/coordinators/model builders and the designers to work together to create a next release. Information from different people in different forms must be collected aiming at a final good design. A problem occurring during product development is, how to know which people to contact for what kind of information, but that is only one. During all of the process concurrent engineering, particularly for the needs of complex very large scaled integrated system design, needs to keep everything in order and on track, while allowing people to work on many different aspects of the project at the same time with differing authorizations of control from anywhere at any-time.

For the purpose of the following discussion, need to say that we call our system a "Computer Integrated Design Control System and Method" because it encompasses the ability to integrate CIM, EDA, PDM and PIM and because it has the modularity making it possible to fulfill these needs in a concurrent engineering environment particularly useful to the design of complex very large scaled integrated systems as employed in a computer system itself. The making of these systems is a worldwide task requiring the work of many engineers, whether they be employed by the manufacturer or by a vendor, working in parallel on many complete parts or circuits which are sub-parts of these parts. So as part of our development, we reviewed the situation and found that no-one that we have found is able to approach the creation of "Computer Integrated Design Control System" like ours or employ the methods needed for our environment. Our methods are modular and fulfill specific functions, and yet make it possible to integrate them within a complete "Computer Integrated Design Control System".

A patent literature review, especially one done with retrospective hindsight after understanding our own system and method of using our "Computer Integrated Design Control System" will show, among certainly others, aspects of DMS systems which somewhat approach some aspect of our own design, but are lacking in important respects. For instance, after review of our detailed description, one will come to appreciate that in modern data processing systems the need often arises (as we provide) to aggregate disparate data objects into a cohesive collection. These data objects may reside at various levels of completion, spanning multiple versions and/or repositories in a hierarchical, multi-tiered data management system. Additionally, these data aggregations may need to be hierarchical themselves, in order to enable the creation of large groupings of data with varying levels of granularity for the data included therein. In such a data management system, the end-users of the data aggregates are not necessarily the "owners" of all or any of the data objects comprising the data aggregate, but they have a need to manage the particular collection. Management of a data aggregation may include creating the aggregation, adding or deleting data objects, moving the aggregation through a hierarchical, multi-tiered data management system and tracking the status of the data aggregation in real-time while maintaining the coherence of the data aggregation. Creation of a data aggregation or the addition of a data object to an existing data aggregate may need to be accomplished within the data management system or via data objects imported into the data management system through application program interfaces for the data management system.

With such a focus, when one reviews the art, one will certainly find, currently, data management systems which provide means for grouping components of a data system to facilitate the retrieval thereof. However, these data management systems are insufficient and lacking because they fail to address the above-referenced need for grouping disparate data items, just to mention one aspect of our own developments.

Another example, U.S. Pat. No. 5,201,047 to Maki et al. (issued Apr. 6, 1993) teaches an attribute based classification and retrieval system wherein it is unnecessary to implement an artificial code for indexing classifications. The patent teaches a method for defining unique, user-determined attributes for storing data which are capable of being readily augmented without necessitating the modification of the underlying query used for retrieval thereof. However, the Maki et al. patent requires that the data items being grouped

share at least one common attribute to enable the grouping, and therefore fails to address the problems of managing data aggregates formed from disparate and unrelated data objects.

Other data management systems address the creation of data aggregates coupled to particular processes implemented in the data system. For example, U.S. Pat. No. 5,321,605 to Chapman et al. (issued Jun. 14, 1994) teaches the creation of a Bill of Resources table which represents the resources consumed in the performance of a given process. Attribute tables for the given resources are utilized to determine whether an additional processes which will consume some or all of the resources of a given process can be initiated. The patent to Chapman et al., requires that each process to be initiated have a particular Bill of Resources aggregate associated therewith. This tightly coupled construct does not permit the creation of data aggregates not related to a particular process implemented in the data management system. Furthermore, since a process must be contemplated in order to create a Bill of Resources table, Chapman et al. do not permit the creation of aggregates without foreknowledge of the process that requires the resource. Thus, in a manner similar to that described for Maki et al., Chapman et al. require that a relationship between the elements exist prior to the formation of the Bill of Resources grouping.

Also, unrelated DMS systems are known which are used for hardware implementations which enable related data in a computer memory, storage or I/O subsystem to be physically grouped in proximity to other such data so as to improve hardware performance, application performance, and/or to solve memory management issues are known. For example, U.S. Pat. No. 5,418,949 to Suzuki (issued May 23, 1995) teaches a file storage management system for a database which achieves a high level of clustering on a given page and teaches loading related data from a secondary storage unit at high speed. The patent uses map files including a metamap file for defining page to page relations of data. These hardware implementations are not related to the present invention, as they involve the management of the physical contents of a data object rather than the management of aggregations of data objects as we perform the methods of our present invention. It is contemplated, however, that such known hardware techniques may be implemented in a system comprising the aggregation management features disclosed herein, thereby further augmenting the overall system efficiency.

During our development process we have viewed the development of others. Even the best of the EDA (electronic design automation) design houses don't have an integrated approach like we have developed.

For the purposes of this background, we will discuss some of the various approaches already used specifically viewing them in light of our own separate developments which we will further elaborate in our detailed description of our invention which follows later in this specification.

In the field of EDA, there are today two preeminent vendors of development software, Cadence Design Systems, Inc. and ViewLogic, Inc. Of course there are others, but these two companies may have a greater range of capability than the others. Also, there are in house systems, such as IBM's ProFrame which we think is unsuitable for use. It will not function well as a stand-alone DM point tool for integration into a foreign framework. But even the largest microelectronic systems are customers of the two named vendors which we will compare. Today, a DCS, it will be seen, without our invention, would require fitting together

pieces of disparate systems which don't interact, and even such a combination would not achieve our desirable results.

For the purposes of comparison, after our own description of our environment, our "Computer Integrated Design Control System", we will discuss the features of the Cadence Team Design Manager Version 4.4 and ViewLogic's View-Data in Sections which compare with and refer to the Sections of our own preferred "Computer Integrated Design Control System" as set forth at the beginning of our detailed description of our invention. This comparison will show the shortcomings of these prior systems, as well as some changes which could be made to these prior systems to allow them to improve performance in our concurrent engineering environment by taking advantage of aspects of our own development as alternative embodiments of our invention.

Historically many attempts have been made to collect or group objects together in order to solve typical data management problems. These problems may include identifying all of the files used to create a model, or grouping files together to facilitate transport through a medium. The intent is usually to ensure the group remains together for a specified period of time.

The most common method in use today is to create a listing of files commonly referred to as a Bill of Materials. Many commercial products permit creation of such a BOM, but these BOM are static list BOM. For example, is one of the members of the BOM disappears or gets changed, the user is unaware that the original data set used to create the BOM is no longer valid.

We have created a new process which we call an Aggregation Manager which can be used in Bill of Materials applications but which overcomes prior disadvantages and also one which can be used in our Computer Integrated Design Control System.

SUMMARY OF THE INVENTION

In accordance with our invention we provide a method for computerized design automation, comprising, accessing a file and database management system for managing a plurality of projects as a design control system, organizing data repository for each project for data records and a control repository comprising a common access interface and one or more databases, said control repository communicating with users of said design control system for fulfilling requests of a user and the data repositories of said data management control system through a plurality of managers, each manager performing a unique function. And within this environment we provide application support whereby users combine selected ones of said managers for supporting an computerized design automation application environment suitable for multiple users of a user community located at workstations anywhere in the world accessible via a network, an intranet, extranet or via the internet.

Thus it will be seen that our invention relates to storing, moving, retrieving and managing data in a system comprised of one or more shared public libraries interacting with one or more private libraries arranged in a client server environment. Elements of the system may exist on a homogeneous computer platform, or the elements may be scattered across multiple platforms in a heterogeneous environment. The Design Control System incorporates processes for hardware design, software development, manufacturing, inventory tracking, or any related field which necessitates execution of repetitive tasks against multiple iterations of data in a quality controlled environment and our invention enables sharing of libraries in this environment for concurrent engineering.

We provide with these applications various systems, methods and processes for data management particularly suited for use with a data management system having shared libraries for concurrent engineering from locations anywhere in the world, along with an management systems allowing promotion of multiple design BOMs through various levels of development, while handing multiple problems, multiple releases and multiple parts control for computer integrated design control within our data management system for releases, and file and database management.

Our invention provides a design control system usable in a concurrent engineering process which can cooperate in a distributed environment worldwide to enable a design to be processed with many concurrent engineering people and processes. The system we employ uses a data management control program tangibly embodying a program of instructions executable by a supporting machine environment for performing method steps by an aggregation manager of a data management system having a library organization which receives a request of a user initiated from said displaced client screen and fulfills the request by providing result via our data management system's aggregation manager.

Our invention provides an improved way to make or import a model, and provides a dynamic way to track the model during its course through its design phase. We provide a way to track the BOM.

In order to make a common model, we display for creation of a model one or more control screen sections as part of a control panel input screen allowing creation of a model by interactive user activity, by importing a file listing, by searching of a library of files in said data management system and importing a located file, or by use of an application program interface with a collection of model management utilities. Our sections of said control screen panel include:

- (a) a display screen section displaying a first field representing the name of an anchor name field of a model which is identical to the name of a data object which is serving as a named anchor;
- (b) a display screen section displaying a second field representing a library where said named anchor resides;
- (c) a display screen section displaying a third field representing the type of data object identified by said anchor name;
- (d) a display screen section displaying a fourth field representing user entries for the version of said named anchor;
- (e) a display screen section displaying a fifth field representing user entries for the level of said named anchor for use by a user or a third party tool for creating, modifying or deleting an aggregate collection of data objects, encompassing those used for items that are identified, tabulated, tracked, validated and invalidated, and promoted, as are bills of materials, by said data management system; and
- (f) a display screen section displaying a sixth field representing user entries for the status of said named anchor.

Our model thus consists of one anchor and one or more associated components, each of which is a data object in said data management system. This means that our components, can belong to any level and version of any library in said data management system and said components are not restricted to the same library, level and version as the anchor, and our components can and do comprise multiple data types, including data generated by tools of said data man-

agement system and third party tools. Thus we further provide that each component is labeled as an input or an output of its associated anchor. Thus we provide that each one component may be an anchor to another different model, and when such a component is an anchor to another different model, said different model consists of said said such component acting as one anchor and further consisting of one or more associated components each of which is a data object in said data management system. In accordance with our invention our components components can belong to any level and version of any library in said data management system and our components are not restricted to the same library, level and version as the anchor, and our components can comprise multiple data types, including data generated by tools of said data management system and third party tools.

Each of our components has field identifiers like those of our anchor and each component is also labeled as an input or an output of its associated anchor. Each one component may be an anchor to still another different model, with each component being labeled as an input or output in relation to its anchor file. All components of a model are either static and thus does not move through said data management system but is tracked by the system or dynamic and moves through said data management system with its associated model as part of an action of promoting a model when a model is promoted, a dynamic member being labeled as an input or an output with respect to its associated anchor, while both anchors and components may be labeled as static.

With these facilities, concurrent engineering is enhanced, and after creation of a model, thereafter, our system provides continuously tracking the created model while allowing a user to modify it by adding components, deleting components, changing the status or deleting said created model, and allowing promotion of a model in our data processing system through the libraries of our data processing system.

This, along with many other changes have been made as detailed in the description of our invention which follows.

BRIEF DESCRIPTION OF THE DRAWINGS

The subject matter which is regarded as the invention is particularly pointed out and distinctly claimed in the concluding portion of the specification. The invention, however, both as to organization and method of practice, together with further objects and advantages thereof, may best be understood by reference to the following description taken in connection with the accompanying drawings in which:

FIG. 1 illustrates a prior art system in which our present system can operate by changes made to the database and design control system, in accordance with our detailed description.

FIG. 2 illustrates our preferred embodiment's data entry.

FIG. 3 illustrates our preferred Design Control System Level Structure;

FIG. 4 illustrates our preferred Design Control System Level Structure with Versions;

FIG. 5 (illustrated in parts FIG. 5a and 5b) illustrates our preferred Design Control System Library Search Examples;

FIG. 6 illustrates our preferred Mechanism for Update Locks;

FIG. 7 (illustrated in parts FIG. 7a and 7b) illustrates our preferred Promotion Mechanism;

FIG. 8 (illustrated in parts FIG. 8a and 8b) illustrates our preferred Design Fix Management and EC Control;

FIG. 9 illustrates our preferred DCS Using an Actor/Object Environment; and

FIG. 10 illustrates our preferred Example of Location Independent Data Tracking;

FIGS. 11, 11a-11d describes the QRFILDEL Process.

FIG. 12 illustrates the overall diagram of the Promote Process.

FIG. 13 depicts a data entry screen for initiating a Promote.

FIGS. 14a thru 14f describes the algorithm for Promote Foreground Processing.

FIG. 15a thru 15e describes the algorithm for Promote Background Processing.

FIG. 16 illustrates the Overall Structure of our Design Control System's Data Management facilities.

FIG. 17 illustrates the Control Repository.

FIG. 18 illustrates the Data Repository.

FIG. 19 illustrates the Inverted Tree Library Structure

FIG. 20 illustrates the Library Structure File.

DETAILED DESCRIPTION OF THE INVENTION

Overview (Section 1.0)

In order to introduce our Design Control System we will describe it as it can be applied to development of complex circuit design and development projects such as microprocessor design projects. The implementation of our Design Control System can be implemented in a variety of ways using many computing platforms as is suitable for a concurrent engineering project. While we will describe our preferred embodiment, it should be recognized that with this teaching all or part of our exact implementation of user interfaces, methods, features, properties, characteristics and attributes may vary depending on the platform chosen and the surrounding design system. All of these variances will nevertheless employ those routines which implement our processes and which meet our requirements.

Platform (Section 1.1)

The Design Control System (DCS) in our preferred embodiment, even though it can be implemented with other platforms, runs on a network of RS/6000's (workstation class "personal" computers) with an AIX operating system arranged in a Client-Server fashion. Each client and server in our preferred embodiment, is able to implement cross platform code via interpretation, and thus can implement programs written in cross platform languages like Java and VRML. In such situations, Java can interact with VRML by describing extension modes, acting as scripts, and describing the actions and interactions of VRML objects.

While more powerful situations are contemplated, the system can be installed in a prior art system, like that described in U.S. Pat. No. 5,333,312. Thus, as we show in FIG. 1, the prior art system of the earlier patent, can be employed in this application, by providing the system with new programs. However, such a system, as illustrated by FIG. 1 will be a data processing system 8, which may include a plurality of networks, such as Local Area Networks (LAN), 10 and 32, each of which preferably includes a plurality of individual computers 12 and 30 (which may be RS/6000 workstations or powerful PCs such as the IBM Aptiva's. As common in such data processing systems, each computer may be coupled to a storage device 14 and/or a printer/output device 16. One or more such storage devices 14 may be utilized, to store applications or resource objects

which may be periodically accessed by a user within the data processing system 8. As we have said the system is provided with a repository, illustrated by main frame/server computer 18, which may be coupled to the Local Area Network 10 by means of communications links 22, and also to storage devices 20 which serve as remote storage for the LAN 10. Similarly, the LAN 10 may be coupled via communications links 24 supporting TCP/IP through a subsystem control unit/communications controller 26 and communications link 34 to a gateway server 28. Gateway server 28 is preferably an individual computer which serves to link the LAN 32 to LAN 10. The main system can be located anywhere in the world, and remotely from the various servers and clients coupled to it over communications links. The main system can accommodate hundreds of users making requests to the centralized repository (a large server 18, such as one of IBM's S/390 platforms or IBM's RISC System/6000 Scalable POWERparallel Systems (SP) platform for design control information: (AIX, S/390, RS/6000, RISC System/6000 and Scalable POWERparallel Systems are trademarks of International Business Machines Corporation, Armonk, N.Y.)

Since this repository 18 (a large server and its associated storage) is critical to the entire design team, it has the ability to remain available if a single server fails. In addition, the data is secured via a backup or archiving mechanism performed on a regular basis. Our DCS has important performance characteristics. It can handle a distributed computing environment with data being transmitted over LANs and telephone lines linking distant locations in real time. Users at one site experience no noticeable delays accessing data physically located at another site. Due to the complexity of the design, maximum throughput is attained by transferring only the control data necessary to carry out the specific task. For large projects design control information can be physically segregated by library, version and level to minimize the bottleneck caused by too many users accessing the same physical server. In the case of the design data, the physical data is tracked via pointers whenever possible, so as to minimize the amount of file movement between servers. Although, the "official" control information is centralized in one place, the DCS permits certain data to be cached locally on the users machine to improve performance by reducing traffic to the Design Control Repository. For example, much of the control information for private libraries can be cached locally in order to maximize performance for private library accesses. For public libraries, the DCS allows the user to take "snapshots" of a library in which the image of the library is refreshed locally. The user continues to work with his local image of the library until he deems it necessary to refresh the image. The amount of control data that is actually cached is dependant on the environment and the actual implementation. Many of the performance issues are discussed further in the Sections to which they pertain.

Libraries and Design Control Repository (Section 1.2)

The Design Control System has two important components. The Design Control Repository contains the control information for all components of the design. This includes such things as the names of all the pieces, the type of data, the level, the version, the owner, and any results which are deemed quality control records. These results indicate the "degree of goodness" of the design component and they are used by the DCS to make decisions regarding the type of actions which can be performed on a piece of data. This repository can be and is preferably implemented in the form of a database (relational, object oriented, etc.) on using a flat-file system. The actual implementation is usually based on the environment.

As we have said, and as illustrated by the machine to person interface depicted by FIG. 2, our program of instructions executable by a supporting machine environment for performing method steps by an aggregation manager of a data management system having a library organization which receives a request of a user initiated from said displayed client screen as illustrated by FIG. 2 and fulfills the request by a providing a result which provides a dynamic way to track a model during its course through its design phase via our data management system's aggregation manager.

In order to make a common model, we display for creation of a model one or more control screen sections which provide our control information components 235, 236, 237, 238, and 239 as part of a control panel input screen allowing creation of a model by interactive user activity, by importing a file listing providing the data of screen sections 235, 236, 237, 238, and 239; by searching of a library of files in said data management system and importing a located file containing the data of screen sections 235, 236, 237, 238, and 239, or by use of an application program interface with a collection of model management utilities which provides the data of screen sections 235, 236, 237, 238, and 239. These data fields of our control screen which when created by a user comprise data entered in the form boxes (a form is a screen section entry field for representing a model) illustrated in FIG. 2, and when retrieved or otherwise obtained by the system by importing a file listing providing the data of screen sections, by searching of a library of files in said data management system and importing a located file containing the data of screen sections, or by use of an application program interface with a collection of model management utilities all provide the data of a control screen panel sections which include:

(a) a display screen section displaying a first field representing the name (235) of an anchor name field of a model which is identical to the name of a data object which is serving as a named anchor;

(b) a display screen section displaying a second field representing a library (236) where said named anchor resides;

(c) a display screen section displaying a third field representing the type (237) of data object identified by said anchor name;

(d) a display screen section displaying a fourth field representing user entries for the version (238) of said named anchor;

(e) a display screen section displaying a fifth field representing user entries for the level (239) of said named anchor for use by a user or a third party tool for creating, modifying or deleting an aggregate collection of data objects, encompassing those used for items that are identified, tabulated, tracked, validated and invalidated, and promoted, as are bills of materials, by said data management system.

Furthermore, while, as in the other cases for entry section fields, the same screen does not have to, but can, display an additional field which displays status information. Thus, as illustrated by FIG. 2, the system provides a display screen section displaying a sixth field representing user entries for the status of said named anchor. Now each field can be display separately and various combinations can be made, but all fields are provided by and used by our system. At any time, the entire model schema can be displayed, as it is in the field 240, which displays several models names, as well as their anchor, type, library, version, level and status (which is dynamically tracked by our system).

Our model thus consists of one anchor (with a name 235) and one or more associated components, each of which is a data object in said data management system. This means that our components can belong to any level and version of any library in said data management system and said components are not restricted to the same library, level and version as the anchor, and our components can and do comprise multiple data types, including data generated by tools of said data management system and third party tools.

Now once a model is created or otherwise identified, it becomes part of our system. Indeed the second component is our Design Libraries. They hold the actual pieces of design under the control of the system. There is no limit to the number of libraries under the management of the Design Control Repository, and hierarchical designs are allowed to traverse through multiple libraries. The libraries are managed by Data Managers (Librarians) who are members of the design team. All major facets of the libraries are programmable so they can be tailored to the needs of the design group they service. Certain design groups require more data control than others, so the flexibility exists to widely vary the degree of data control. Libraries are categorized as Public or Private. Both can be shared, but the main difference is that a private library is managed by the actual designer. It's used to hold his daily updates and often will have no formal control. The DCS achieves this by defaulting all control information to a simple non-restrictive form. For example, any designer can create private libraries on their own. They automatically become the owner and have the right to make additional designers "backup" owners. As the owner they can edit, save, modify, or delete any data in their library. The DCS automatically establishes all the proper AFS and AIX permissions. Owners of private libraries control who can access their data with the system accommodating the use of default "access groups" (such as AFS groups) so the designer doesn't have to enter the userids of all his team members each time he creates a new library. Since Private Libraries are considered working areas, data control checks are minimized in order to maximize performance. For example, when a new data element is created, the DCS does not check the Control Repository to make sure the owner has the proper authorities, locks, etc.. Instead, a designer is permitted to work in a completely unrestricted fashion in his own work space. All controls are placed on public libraries. The only control checking required is to ensure there are no data conflicts within the Private Library. It is acceptable for two Private Libraries to contain the same design data, so no checks across libraries are done. Public Libraries are the official project data repositories. All data delivered to external customers comes from Public Libraries. Public Libraries are overseen by Data Managers who configure the libraries with varying degrees of control. Typically the libraries are organized with a level structure whereby the lowest levels have the least amount control. Control gets more stringent as the levels increase, and the highest level denotes data released to manufacturing. Almost every attribute concerning data integrity is programmable by the Data Manager. Through a Data Manager Utility, they configure the structure (the number of levels and versions, including the connections between them), the various authorities, the required criteria to enter each level, and the types of Library Controlled Processes required at each level. The system can handle numerous public libraries, and each public library can service unlimited users. In accordance with our preferred embodiment of our DCS architecture we provide an Automated Library Machine (ALM). More than merely a repository for data, the ALM is a userid capable of

accepting, executing and dispatching tasks without any human intervention. This enables the designers to make requests of the ALM to promote data or run library processes without the need for a Data Manager to process it.

In order to improve throughput, the ALM can dispatch parallel tasks if the operating system (i.e. AFS) supports it and the situation allows it.

This concept improves efficiency, and increases security, since the ALM is the only user that requires writable permissions to the data repositories. The physical location of the data residing in Public Libraries is determined by the Data Manager. The DCS along with the Data Manager (and his alternates) are the only means of writing data into or removing data from these physical locations. As a means of safety, the Data Manager does have the ability to access and overwrite data in these physical locations without using the DCS (i.e. thru the OS). This is necessary in the unlikely event the control information gets out of sync with the physical data, and the Data Manager has to manually complete a transaction. Physical locations are defined through the Data Manager Utility for setting up Public Libraries. More details on this are available in the Data Manager User Interface Section 15.

Data Types (Section 1.3)

Data may be identified by a filename (anchor name 235) and a filetype (236). The DCS automatically segregates all data by "type". Types are very useful to associate a piece of data with a tool or process. For example, UNIX/AIX uses extensions to qualify data such as using a ".ps" extension to denote a postscript file. The Cadence Design Management System uses Cell Views to segregate the various types of data within a particular Cell (design component). This segregation is a fundamental building block to Design Control Systems since certain types of data require more design control than other types. Our DCS allows each individual type to be controlled on a level and version basis within a library. The DCS is capable of tracking any data type from any point tool, even third party vendors.

Levels (Section 1.4)

Each Public Library consists of n levels which are established by the Data Manager. The naming of the levels (239) are arbitrary, but each denotes a degree of quality of the design. Data moves into and out of levels via a "promotion" mechanism. There are two types of levels in the DCS, Engineering (or working) and Release Levels.

FIG. 3 shows a typical level structure with 3 Engineering Levels denoted E1, E2 and E3, two main Release Levels denoted R1 and R2, a Sideways Release Level S1, and a Fast Path Stream consisting of F21 and F22. Data can be promoted into E1, F21, E3 and S1 from outside of the library, but it can only enter R2 from E3. E1, E2 and E3 are arranged in a serial fashion. The normal promotion path is for data to enter E1 (the least controlled level) and migrate up through E2, E3 and finally into R2 (the most tightly controlled level). The external paths into F21 and E3 are known as "fast paths" and exist to accommodate emergency updates to pieces of design residing at the higher levels. There are two different types of fast path arrangements:

Fast Path Entry means there is no fast path level associated with the Engineering level, just a "doorway" through which data can enter. Level E3 is an example of this where the user simply promotes data from the private library into E3. The DCS will run any pre-processes defined at E3, but any criteria that would normally be necessary to traverse through E1 and E2 is bypassed.

Fast Path Levels are staging areas where data is promoted into, and promoted through, in order to reach the target Engineering Level. There can be any number of Fast Path levels for any given Engineering Level. If there's more than 1, it's known as a Fast Path Stream since the data must migrate through all the Fast Path Levels before reaching the Engineering Level. F21 and F22 constitute a stream, which could've contained more than 2 levels. We have provided at least one level to provide an area where all the processing normally run at the E1 and E2 levels can be run to ensure that the fast path data meets all the same criteria.

Release Levels are handled in a different manner. R1 is the oldest release level and it's frozen, which means its contents can't be updated any longer. It contains a static snapshot of a design delivered to an external customer. R2 is now the active Release Level which is the destination of any data promoted from E3. The Data Manager programs the connection of E1 to E2 to E3 to Rn. The DCS automatically freezes the previous Release Level and connects E3 to the new Release Level whenever the Data Manager creates a new one. Unlike main Release Levels, Sideways Release Levels are always active and there can be n Sideways Levels for each Release Level. The purpose of the Sideways Levels is to hold post tape-out updates such as microcode patches to hardware under test. Since the Release Level corresponding to that level of hardware is probably frozen, and a new iteration of design is propagating through the Engineering Levels, the only path into a Sideways level is directly from a Private Library. The Data Manager has the ability to reconfigure the Engineering Levels at any time based on these rules:

The connections between levels can be changed at any time. (i.e. E→E2→E3 can be changed to E1→E3→E2.)

A level can be removed as long as no data resides in that level.

A level can be added at any time.

The Data Manager can create a new Release Level at any time. Existing frozen Release Levels can be removed as long as no data resides in that level. A frozen level can become an active level again if no data resides in the current active Release Level. The DCS performs a "thaw", a step which removes the current Release Level (R2) and connects the previous level (R1) to E3. As shown in FIG. 3, the DCS supports the normal promotion path to E1 as well as "fast paths" into E2 and E3. The following minimum checks are performed at all entry points:

The owner attempting to send data to a Public Library must possess the update lock. If no lock exists, the sender obtains the lock by default. If another user has the lock and the sender is a surrogate, he can obtain the lock (the system immediately notifies the original owner). If the sender is not a surrogate, the action is halted, until ownership is properly transferred.

If the level to which the data is being promoted to has any entry criteria, it is checked to ensure the data passes the criteria.

Versions (Section 1.5)

Each public library consists of n versions which are defined by the Data Manager. The concept of versions exist to support parallel design efforts. All versions have the same Engineering (Working) Levels, but have different Release Levels depending on the frequency of tape-outs for that version. Data in separate versions is permitted to traverse the levels at independent rates. For example, if a piece of design

has 2 versions, 1 version may exist at E1 while the other version exists at E3. FIG. 4 is an extension of FIG. 3 in which library structure has been expanded to show 3 versions, V1, V2 and V3. In theory there's no limit to the number of versions just as there's no limit to the number of levels. Versions can be independent or dependent. Independent versions are isolated and must ultimately contain the entire set of design components. Dependent versions are based on previous versions (which the Data Manager specifies when creating a new version). By supporting the concept of dependent versions, only the incremental data necessary for a new design variation needs to be librated in the new version. The Library Search mechanism will be able to construct a complete design Bill of Materials by picking up data from both versions.

Library Search (Section 1.6)

Our preferred embodiment of the DCS provides support for "Library Searches". This allows data, which is used in multiple iterations of the design, to exist in only one place. In other words, if a design component is changed, only that component needs to be re-librated at a lower level. A full model can still be constructed by starting the search at the lowest level where the component is known to exist. The library search mechanism will pick up the latest pieces at the lowest level, then search through the next highest level to pick up more pieces, and so on until it reaches the highest level where all components reside. In addition to searching through levels, the mechanism also searches through versions. The user provides a starting library level, version and either a single type or a list of types. If the version is based on a previous version, and all the necessary design components can't be located in the starting version, the mechanism searches the previous version based on the following two rules:

1. If the search begins at an Engineering Level in one version, it resumes at the same Engineering Level (not the lowest level) in the previous version.
2. If the search begins at a Release Level (including a Sideways Level) in one version, it resumes at the latest Release Level in the previous version. This may be older or more recent in time than the released data in the current version.

FIG. 5 shows examples of the Library Search Mechanism. The library search utility is available to designers, Data Managers and third party tools. The interface is both command-line and menu driven to accommodate any environment. In addition to the required parameters of type, level and version, the user has the option of specifying the name of a data object. These additional options exist:

Noacc

This allows the utility to use a temporary cached copy of the search order information for performance reasons. Since this information may be obsolete, the absence of the option results in the actual Design Control Repository being accessed and the search performed from within it.

File

Write the results into an external file.

Various Sorts

They control the way the output is sorted and displayed.

Nosearch

Only list data found at the starting level.

First/All

Indicates whether to include all existences of a particular design component or only the first one in the search order.

Select

Presents a selection list of all candidates so the user can choose those of interest.

Noversion

Prevents the search from tracing back across version boundaries.

Levels

Displays the search order based on the existing level structure.

Versions

Displays the search order based on the existing version structure.

Locks (Section 1.7)

In order to properly control shared data, the DCS supports several types of locking mechanisms. Two of the locks exist to control groupings of files that may comprise a model build. These are known as move-and-overlay-locks. The user can set one of these locks using a utility which allows him to control the scope of the lock based on certain fields. The user can enter specific data or a wildcard, indicating "ALL", for

Name of Design Components

Type of Design Components

Level of Design Components

Version of Design Components

Library Name

By specifying only a library name and four wildcards, the user is requesting that all data in the library be locked. By filling in all five entries, a specific design component will be locked. Various degree of locking exist in between those extremes.

If the information corresponds to a Bill of Materials (BOM) and the user wants to set the lock on the entire BOM, a BOM Flag will exist allowing him to specify this action. Regardless of how these fields are filled in, all locks will be set individually so they may be removed individually. A lock does not have to be removed the same way it was set. The user will also specify the type of lock, Move, Overlay, or Update (Ownership). The following definitions exist:

Move Locks mean the data can't be overlaid by the same data at lower levels, nor can it be promoted to a higher level. This provides a method for completely freezing an Engineering Level while a model build or large scale checking run is in progress.

Overlay Locks are a subset of move locks. The data can't be overlaid by the same data from lower levels, but it can be promoted to higher levels.

Update (Ownership) Locks are the means by which a designer takes ownership of a piece of data. Update locks are designed to prevent multiple designers from updating the same design component in an uncontrolled way, thus resulting in data corruption or lost information. There are two types of Update locks, permanent and temporary.

A permanent Update lock exists when the designer specifically requests to own a piece of data. This is done through a utility, and the DCS keeps track of this ownership. Other designers may copy and modify the data in their private libraries, but any attempt to promote that data into the public library will fail, unless the designer is a designated surrogate of the owner. The only way these locks are removed are by the owner resigning the lock or a surrogate assuming the ownership of the data, and the corresponding lock. A temporary Update lock exists to facilitate sharing a piece of data among multiple designers. The user can either request a

temporary Update lock in advance (i.e. when he begins editing the data), or he can wait until he initiates the promote into the public library. The DCS will first check to see if anyone has a permanent Update lock, and if so, it will only allow the promotion to continue if the user is a designated surrogate. If nobody has a permanent Update lock, then the DCS will issue a temporary Update lock for the time the data remains "en route" to the final promote destination. Once it arrives safely, the temporary Update lock is removed and the data can be claimed for ownership by someone else. Surrogates are "alternate" owners of data. For example, a project may be arranged such that each piece of design is owned by a primary designer, but also has a backup owner (designer) to take over the design during vacations, emergencies, etc.. In this case, the owner can tell the DCS that the backup designer should be a surrogate, thus giving him the right to take ownership of a design component. The surrogate can either use the locking utility to specifically take ownership prior to making any updates, or he can wait until he initiates a promotion. The DCS will check to see if the design component is currently owned, and if so, check to see if the user is a defined surrogate. If both are true, it will give the user the chance to "take ownership" and allow the promote to continue. The original owner would be notified that his surrogate has taken ownership. FIG. 6 illustrates the lock mechanisms for Update locks.

Bill of Materials Tracker (Section 1.8)

The DCS has a built-in Bill of Materials (BOM) Tracker to facilitate tracking many design components in large projects. The main objective of the BOM Tracker is to group certain design components to make it easier to promote them through the library and track their synchronization. This is crucial for data sets that contain some source and some derived files from that source. The following features exist in the BOM Tracker:

It supports automatic data grouping, based on the design component name, with the notion of required and optional data types. One example might be a grouping which consists of a graphical symbol denoting the I/O of a design component, the corresponding piece of entity VHDL and the architectural VHDL. Any changes made to the symbol should be reflected in the entity, so the entity would be required. A change may also be made to the architecture, but it's not always necessary, so the architectural VHDL would be optional. When a promote is initiated to a public library, or between levels of a public library, the DCS checks to see whether a data grouping is defined for the data type being promoted. If so, then all required data types are checked to ensure they exist. In addition, any optional data types are checked for existence and they are also picked up. The entire grouping is promoted to the target level. If a required data type does not exist, the promotion fails. Automatic data groups are programmed into the DCS by the Data Manager. Since they are BOMs, all rules of BOM tracking, invalidation and promotion exist for the members of the grouping.

BOMs are used for two main reasons. First they are used to group many smaller pieces of data into larger more manageable chunks to facilitate movement through the library and increase data integrity by reducing the risk of data getting out of sync. The other main reason is to track the components of a model (i.e. simulation, timing, noise analysis, etc.). The DCS offers a very flexible user interface for creating BOMs in order to satisfy the various scenarios. The user can manually create BOMs by selecting pieces of design

interactively, filling in search criteria and initiating a library search, or importing a simple text list. In addition, an API exists for point tools to create a BOM listing and pass it into the DCS.

The power of the BOM Tracker is augmented with our automatic invalidation routine. Once a BOM is created, the DCS constantly monitors for a change to the BOM. If any member is overlaid or deleted, a notification is sent to the owner of the BOM indicating that the BOM is no longer valid. The owner can continue to work with his model, but he is aware that he's no longer using valid data. Even though a BOM is invalid, it can still be moved through the library. This accommodates the occasion where a piece of a model had a relatively insignificant change. If the model builder deems it unnecessary to re-build the model, this feature allows him to continue his work and even move the BOM through the library.

Status on BOMs is and should be accessible in two ways. The first is by automatic notification (e.g. e-mail) to the owner as soon as a BOM is invalidated. The second is by means of displaying the BOM either interactively or in report form. This listing shows the overall status of the BOM, and all members of the BOM with their individual status.

The BOM Tracker also supports the concept of a "support" object. This can be a design component, a piece of information, documentation, etc., that can be associated and promoted with a BOM but never causes BOM invalidation.

BOMs are hierarchical in nature and a BOM can be nested within a larger BOM. Whenever a piece of data is overlaid or deleted, the DCS looks to see if that piece belonged to a BOM. If so, it immediately checks to see if the BOM belongs to other BOMs. It recursively checks all BOMs it encounters until it's at the top of the hierarchy. All BOMs found will be invalidated (if they are currently valid) and the owners notified.

BOMs support move and overlay locks. The user can set a move or overlay lock on a BOM, and the DCS will set individual locks on all the members. If a member is a BOM, all of its members will receive individual locks. These locks can be removed by using the main lock utility and specifying the top-level BOM or filling in the desired fields to individually reset locks.

The DCS supports the concept of a BOM promote, which means the user can request that all the contents of the BOM be promoted simultaneously. This increases data integrity by helping to ensure a matching set of design data traverse through the library in sync.

BOMs can contain members who reside at different levels, different versions and even different libraries. The DCS will only promote those members which exist in the current library, and reside in an Engineering Level below the target level. If a member exists in a different version and is also below the target level, it will also be promoted.

There is separate authorizations for creating and promoting BOMs. This is set up by the Data Manager, so they can have complete flexibility in controlling who can create and move BOMs.

Promotion Criteria and Promotion Mechanism (Section 1.9)

An important aspect of the DCS is that it provides a method for the design to traverse to different levels of goodness. As the design stabilizes at the higher levels, the

number of pieces which need to be moved and tracked can be very large. The DCS uses the concept of promotion criteria and robust mechanisms to first determine what data can be promoted, then carry out the task in an expedient manner. The DCS supports two variations, "move" and "copy", promotes. In a "move" promote, data appears to the user like it only exists at the target level once the promote completes. The user is unable to access the copy that existed at the previous level. For example, if a design component is at level E2 and the user promotes it to E3, when the promote is finished and the user refreshes his image of the library, he sees the data at E3 only. In a "copy" promote, the data still appears at the previous level. The user can access it at either location. As new iterations of the same design component are promoted into a level, the old component is not truly overlaid. It is moved off to the side so it can be restored in an emergency. Promotion criteria usually exists in the form of library process or pseudo-process results, but in general it can be any condition that must be met by the object(s) being promoted. It is defined by the Data Manager and can exist for any design component at any level and version. Certain design components don't undergo any formal checking or evaluation in the design process, so they may never have any promotion criteria. Other pieces may undergo the majority of checking so they may have lots of criteria. The objective of the DCS is to track actual results for each design component and use the promotion criteria to determine if the design can attain the next level of goodness. When a design component is overlaid or deleted, all corresponding results are deleted too. The DCS supports an emergency override mechanism which allows the Data Manager to promote data which does not meet the criteria. Invoking an emergency override cause a log entry to be written indicating criteria has been bypassed. The Data Manager determines which results are necessary for which types of design at each Engineering and Release Level. These results may get recorded through "library controlled" or "external" processing. At the time the promote is initiated (whether it be against individual design components or BOMs), the mechanism illustrated by FIG. 7a and FIG. 7b is invoked to determine what pieces should be promoted. There are three types of promote transactions.

1. Promotion of an Individual Design Component
2. Promotion of a Group of loosely-coupled Design Components
3. Promotion of a Group of tightly-coupled Design Components (i.e. BOMs)

Basically, the same mechanism is employed in all three cases, but cases 2 and 3 require additional optimization for high performance. In case 1, each step in the mechanism is executed once and the promotion either succeeds or fails. Case 2 is initiated by a user selecting a group of objects to be promoted. They may or may not have any relation to each other. In this case some optimization is done, but each object is basically treated as if it were initiated as an individual promote. For example, the authority check only needs to be done once since the same user is requesting the promotion for all the objects. However, since each object can have unique locks, criteria, processes defined, etc., most of the steps need to be repeated for each object. Case 3 is the most complicated because the DCS offers a great deal of flexibility. The actual implementation is dependent on the platform of the DCS and the type of control mechanism in place (file-based, object oriented database, relational database, etc.). If the user community wants to eliminate flexibility in return for increased performance, the DCS can enforce rules such as no library processing allowed for members of a

BOM. In this scenario, the entire algorithm would be executed on the BOM itself to ensure the proper authority is in place, it meets the promotion criteria, and any processing that's defined is executed. However, each member could bypass some of the checks thus saving a significant amount of time. If the user community opts for flexibility, some optimization can still be performed. For example, if a BOM contains 10 members and the mechanism calls for five checks on each member, there doesn't need to be 50 requests for information. Depending on the platform, it may be optimal to either make one large request for each member (ten total requests) and obtain all five pieces of information in the request. In other cases it may be optimal to initiate a request for a piece of information, but solicit it on behalf of all ten members (five total requests). Since these BOMs can be extremely large, the various kinds of optimizations and trade-offs between flexibility and performance determine the exact implementation. As a convenience feature, the DCS supports a multiple promote feature which allows the user to request a promote through multiple levels. For each level the promotion mechanism is followed as stated above. For example, when initiating a promote, the user can specify to move data from E1 to E3 with a single invocation. However, the DCS will internally break it into two separate promotes with the full mechanism being run for the E1 to E2 promote, then again for the E2 to E3 promote.

Library Controlled Processing (Section 1.10)

The concept of Library Controlled Processing allows tasks to be launched from a public library, against one or more design components, with the results being recorded against the components. This is an automated method to ensure that tasks, and checks deemed critical to the level of design are run and not overlooked. Since some of these tasks could be third party tools, the actual implementation can vary in sophistication. In its simplest form, Library Controlled Processing consists of the following constituent parts:

Foreground Processing:

This is the conduit by which the user enters any information required to run the tool. Menus may be presented or the user may interact in some other way.

Pre-Processing:

This refers to a library controlled process that is launched prior to the data being promoted to the target level. The process must finish and complete successfully, based on the promotion criteria of that process, if the promote is to continue. For example, if a pre-process is defined at level E2, then when the promote to E2 initiates, the process is launched and the promote "suspends" until the process completes. Once it finishes, the result is compared against the criteria to ensure it's satisfactory. The promote then resumes.

Post-Processing:

This refers to a library controlled process that is launched after the data arrives at the target level. The results of the process are used as promotion criteria to the next level.

Designer Initiated Library Processes (DILP):

This is very similar to a post process, but instead of the DCS launching the process, it's manually launched by the designer. DILPs usually exist to retry Post-Processes which failed. This eliminates the need for the user to re-promote the data just to initiate the processing. If a DILP is used to recover a failing Post-Process, and the DILP is successful, the good result will overwrite the bad result from the Post-Process. Just because DILPs are primarily used to recover failing Post-

Processes, the DCS doesn't make this a restriction. The Data Manager can set up DILPs as stand-alone processes with no corresponding Post-Process. DILPs that exist to recover failed Post-Processes are optional in that they are not counted as required promotion criteria. Stand-alone DILPs can be optional or mandatory, with mandatory DILPs being required to run successfully in order for the data to promote to the next level. The DCS allows the Data Manager to designate which DILPs are mandatory and which are optional.

Level Independent Pseudo Processes:

These are special types of process which are more like process results than actual processes. They exist as a means to record information outside of the scope of results from Library Controlled Processes or External Data Processing. For example, suppose a Library Process exists to run a layout checking program which checks for wiring and ground rule violations. Ultimately the program will return some pass/fail result, such as a return code, which the DCS uses as the process result. The tool may also return other useful information which the designer wants to save, such as the number of wires or cells in the design. Pseudo processes provide a repository for this kind of data. Like DILPs, these can be used as mandatory criteria for promotion, or they can be optional and used solely for information. They can even serve as status indicators for design components progressing through a lengthy process at a particular level. The concept of level independence means the checking program could be run at the E2 level, but the pseudo process results can be stored at E3. In short, the DCS allows a pseudo process to be defined at any level, and it can be set by a process running at the same level, any other level or completely outside of the library. The DCS provides an API for setting level independent pseudo processes. The API can be used by designers, Data Managers or third party tools, and employs a "process search" similar to a library search. This means the API allows the user to specify the name of the process, the data type, level and version. The DCS will use this as a starting level and search for all matching pseudo processes defined at or above this level by following the same library search mechanism as in FIG. 5. A flag also exists to disable the search and set the result for the process specified at that level and version.

Any number of any type of process can be defined by the Data Manager for a given data type at a particular level and version. In addition, processes can be chained together in independent or dependent sequences. In a dependent sequence, each process must complete successfully before the next process in the chain can initiate. For example, when compiling VHDL, the entity must always be compiled prior to the architecture. Thus two compiles could exist as a dependent sequence where the entity is compiled, the result checked, and if successful, the architecture is compiled. In an independent chain, the first process initiates, and when it completes, the next process runs regardless of the outcome of the first process. Processes can also execute using input data other than the object used to initiate the promotion. Using the VHDL compile example, the actual object being promoted could be a simulation BOM which contains that entity and architecture VHDL. The DCS provides a robust system for the Data Manager to define the processes which should be run, and the type of data they should run on. Certain library controlled processes require special resources such as large machines, extra memory capacity,

etc.. Therefore, the DCS allows the Data Manager to specify a particular machine or pool of batch machines where the tasks can execute. Either the task is transferred to the specific machine or a request is queued up in the batch submission system. In the event that a task must run on a completely different platform, the DCS provides hooks to launch a library controlled process from one platform which initiates a task on a different platform (i.e. a mainframe). The results are returned back to the original Automated Library Machine and processed. This Cross-Platform capability allows the DCS to encompass a broad and sophisticated methodology utilizing tools on many platforms. Regardless of how the process is launched, the results must ultimately get recorded within the DCS. To accomplish this, the DCS provides an Application Program Interface (API) through which third party tools can communicate. When the task completes, the API is used to convey the results and the pedigree information back to the DCS. The DCS provides both an interactive means and a report generator to view process results. FIG. 7a and FIG. 7b illustrate the method by which promotions and library controlled processing interact.

External Data Processing (Section 1.11)

External Data Control is very similar to the Designer Initiated Library Process in that the user launches a task against some design component(s). However, unlike DILPs which require that the design components be under the control of a Public Library, this type of processing is done on data in Private Libraries and designer's work spaces. External processing is the mechanism whereby the DCS captures the results of the process along with pedigree information concerning the input data, output data and any necessary software support or execution code. This pedigree information is stored along with the design component for which the designer initiated the process. When the designer promotes that component at a later time, the DCS checks the pedigree information to ensure nothing has changed. It then checks to see if the external processing matches any of the defined library processes which are required for the promote. If so, and the external processing results meet the criteria, the library process results are set (as if the library process just ran automatically) and the promote proceeds. If no matching process can be found, the external results continue to be saved with the design component as they process may match that at a later level. The concept of External Data Processing exists to increase productivity by allowing the designer to save, and later apply, results obtained during the normal course of design rules checking to the "official" results the DCS uses to determine the level of goodness. Overall data integrity can easily be breached if a proper mechanism for calculating pedigree information is not implemented. For this reason it's imperative for the DCS to ensure that all the proper input, output and software data are included in the pedigree information. External Data Processing occurs in two phases. In the first phase, the designer runs some tool or process and if the results are acceptable, he runs a utility to designate the data for external processing. The role of the utility is to create the Pedigree information which contains a listing of the input and output data, the results, and some type of date identification code for each member of the Pedigree and the Pedigree itself. A simple identification code is a cyclic redundancy check. The utility can be independent of or incorporated into the actual third party tool. The second phase consists of librarying the data and the results. The designer invokes a special form of a promote which first does the following:

1. Check the data identification code (i.e. CRC) of all members in the Pedigree

23

2. Check the data identification code of the Pedigree itself.

These 2 steps are designed to ensure the same data used to generate the result is indeed being libaried. The identification code of the Pedigree ensures that the contents of the Pedigree weren't manually altered. From this point on, the normal promotion mechanism in FIG. 7a and FIG. 7b is followed with one exception. The boxes where Foreground, Pre and Post Processing occur are all bypassed. Rather than simply checking existing results to see if they meet criteria, the DCS makes a list of all Pre-processes for the target level and Post processes for the previous level. It then checks the Pedigree information for evidence that equivalent processes were run and achieved acceptable results. If any processes exist in the DCS for which no corresponding Pedigree results exist, or any Pedigree result does not meet the prescribed criteria, the promote fails.

Authorities (Section 1.12)

The DCS permits the Data Manager to establish a wide variety of authorities which gives him great flexibility in managing the library. Each type of authority can be defined very loosely (the user is authorized for all design components, at all levels, in all versions) to very tightly (the user is authorized on an individual design component basis). The utility for granting authorities works in one of two modes:

In one mode the Data Manager is offered a screen in which he can fill in the design component name, type, level, version, user ids, and the type of authority. For any field, except for the user ids, he can default it to "ALL".

In the other mode an authority profile can be called up and executed. An authority profile allows the Data Manager to pre-define the types of authorities for a given type of job. For example, profiles may exist for Designer, Technical Leader, Model Builder, etc.. This information is contained in an editable ASC file in which the Data Manager defines the kinds of authority to varying degrees of restriction. Once the profiles are created, the Data Manager uses this mode to either add/delete users to/from the profile and process the changes within the DCS.

Authorities exist for the following tasks:

Setting Locks (Move, Overlay, Update, ALL)

Promoting design components and/or BOMs into levels (Engineering Levels, Release Level.

Creating BOMs

Initiating Library Processes

Setting Pseudo Process Results

Data Manager GUI User Interface (Section 1.13)

The DCS contains a robust Data Manager interface which is used to "program" the library. It's configured as a series of sub-menus arranged under higher level menus. Each sub-menu has fields to fill in and may employ Predefined Function (PF) keys for additional features. Graphical elements such as cyclic fields, radio buttons, scrollable windows, etc.. may be used to further enhance usability. Utilities exist to:

Define the library properties

The user is afforded a means to enter the path of the repository where the data resides, the userid of the Data Manager and any alternates, the userids of any Automated Library Machines, and whether the library is under Design Fix or Part Number and EC control. If the library is under any type of control, additional entries are made for the data types which should be tracked by Part Number, the data types which should be tracked by

24

Design Fix number, the EC control level, and a field for a generic problem fix number. For any ALMs, the DCS will automatically add the proper authorities (including operating system authorities) to permit the ALM to store data and record results.

Define the structure (levels, versions and their interconnections).

This is the means by which the Data Manager adds and deletes levels and versions. It also enables him to defined the interconnections of the levels, and the dependance of versions on other versions. A minimum interface consists of one screen for level structure and one for version structure. The level structure screen displays the current structure.

Define the types of data which will be under library control.

For all data types known to the DCS, this enables the Data Manager to select those managed in this particular library. The screen displays all known data types in the system with a flag indicating whether it's being tracked by this library. Each data type also has a field for an alternate storage location. This solves the problem caused by certain data types that can be very large. Therefore, problems may arise in trying to store these data types along with the all the other types in a particular level. By specifying an alternate storage location, these large data types can be further segregated.

Manage Library Controlled Processes

For each level, the Data Manager can add, modify or delete processes. For each process information is required about the type of machine it can run on, any necessary arguments, the result criteria, disposition instructions for the output, whether it's dependent on another process, and whether it should be deferred. The DCS provides Process Specific Boilerplates which can be used to manage process configurations for an entire project. Necessary and required information for each process can be programmed into the DCS, so when a Data Manager attempts to define that process to his library, some of the fields appear with default data already filled in. He can override any of the data.

The information for each process can be entered/edited individually on a menu containing all the above fields or a utility exists to load "process groups" which are pre-defined library controlled processes. The Data Manager simply selects a process group and attaches it to the appropriate data type, level and version. The process groups are ASC based files which contain the necessary process information in a prescribed format. They can be created using any ASC editor.

Set up authorities.

See the previous Section 1.12 for details.

Define automatic data groupings (Subset of BOM Tracking)

This enables the Data Manager to define a data group which consists of a master object and member objects. Each member object can be required or optional. For each master object entered, the user must enter a list of member objects with their required/optional flag. In addition, an Erase-To-Level flag exists which determines the outcome of the following scenario: a data group, comprised of optional members, exists at a level. The same data group, without some of the optional members, exists at the next lowest level. Upon promotion of the lower level data group, the DCS will

either erase the members of the upper level data group or leave them, depending on the Erase-To-Level flag. By leaving them in place, it allows members of newer data groups to join with members of older data groups.

Design Fix Tracking (Section 1.14)

136 One of the most powerful aspects of our DCS is provided by the process used to track fixes to design problems. This is accomplished by tightly or loosely coupling the DCS to a problem management database. Typically, a problem is found and entered in the problem tracking database. Once the design components are identified which require updating, the DCS is used to attach the problem number to those design components. Ideally this should be done prior to the design components entering the library, but it can be done as part of the promote. It's often redundant to track all design components with problem numbers, so the DCS can be programmed to only enforce Design Fix Tracking on certain data types. Whenever a promote is initiated, the DCS checks to see if the library is in Design Fix Tracking mode (which means some data types require Fix problem numbers to enter the library), and looks to see if any of the data types included in the promotion are being tracked. For those that are, a screen displays all known problem fix numbers for that design component. The user can select an existing one or add a new one to the list. At this time, the DCS will check to see if the EC control level is being crossed (or bypassed via a fast path promote). If so, it will attempt to associate the problem fix number to an EC identifier. If it can't automatically determine this association, the user is prompted to enter the EC identifier for the selected problem fix number.

If the designer chooses to do the association in advance, a utility exists which allows him to enter a problem fix number or choose a default number. The status is immediately reflected as "working". Once the promotion is initiated the status will switch to "librared". The DCS offers utilities to view or print reports showing which design components exist for a problem or which problems are fixed by a design component. The report generator allows the user to enter the problem number and see which design components are associated to it. Or the design component can be specified to see which problems it fixes. Finally, an EC identifier can be specified and all problem numbers and design components associated with the EC can be displayed.

Part Number/EC Control (Section 1.15)

In addition to tracking design fixes, the DCS can track the design by part number and/or EC. For projects which assign part numbers to various design components, the DCS provides utilities to generate and associate these part numbers to the design components. In addition, the DCS supports Engineering Changes where successive tape-outs are assigned an EC identifier. All design components participating in an EC are associated with the EC identifier. Since part numbers are assigned to specific design components, the DCS uses the links between components design fixes and EC's to track the association of part numbers to ECs. The DCS uses the concept of a PN/EC control level to permit the Data Manager to determine at which level PNs and Design Problem numbers get associated with EC numbers. As design components cross this level, the DCS checks to see whether a problem number or PN exists for the component. If so, and the system is able to determine which EC that number is associated with, it automatically connects the component to the EC. Otherwise, if no EC information can be found, the user is asked to enter it. The rules for Design Fix and EC control are as follows:

One EC can contain multiple Design Fixes;

Any single Design Fix # (number) can only be associated with a single EC;

One design component can have many Design Fix numbers, but they must all belong to the same EC; and Variations of a design component can exist in multiple ECs, but each must have a unique set of Design Fixes.

5 FIG. 8a illustrates a legal example. It shows two EC's where the first contains two design fixes and the second contains a single design fix. There are three design components, of which the one denoted A0 is associated with Design Fix #1 and Design Fix #2. Design component A1 is a different variation of design component A0. The example shows how the two versions of design component A must belong to separate ECs. In FIG. 8b the rules have been violated since design component A1 is associated with Design Fix #2 which belongs to EC #1. The DCS detects this condition and alerts the user to either move Design Fix #2 over to EC #2, or detach design component A1 from Design Fix #2. In addition to tracking all the part number and EC information the DCS is capable of generating a variety of reports including one listing all the part numbers for a given EC. This report can be sent to manufacturing in advance so the foundry can manage their resources.

RAS and Security (Section 1.16)

The DCS is designed in such a manner that provides maximum security for the control data. None of this data is present in simple ASC files residing in a writable repository. All updates to this information must be made through the proper utilities by authorized people. Librared data only exists in repositories where the Data Managers or owners of the data have write permission. This prevents other users from modifying another designer's data outside of the DCS. Nearly continuous availability is achieved by implementing the DCS in the following manner:

If the primary DCS server fails, the system can be brought up on another server with minimal human intervention. The physical locations of all libraries are determined by the Data Manager which permits the data to be strategically located throughout the network to improve availability.

Multiple paths exist to request information from the Control Repository. They provide alternate routes in the event of network or router problems.

Archiving and backing up data is accomplished with the following features:

The Design Control Repository can be archived onto tape or backed up to another repository by the Data Manager as often as deemed necessary. In the event of corruption, this back up copy can be restored into the primary repository.

All libraries can be archived to tape or backed up to alternate repositories defined by the Data Manager as often as deemed appropriate.

The DCS provides a utility which checks to see if a backed-up or archived copy of the Design Control Repository is in sync with a backed up or archived copy of a library. During the archiving procedure, the system assigns unique identification codes (i.e. CRC codes) to each data object. These codes are used during the recovery to ensure the data was not tampered with while dormant on the back-up repository.

The system provides a method for restoring individual data objects from backed-up or archived repositories in the event the data object is deleted from the active library.

GUI User Interface (Section 1.17)

65 The User Interface consists of all the menus, dialog boxes, and screens by which the designers interact with the DCS. They all have the following characteristics in common:

They are user friendly with convenient on-line help.
They share a common look and feel to make it easy for the user to find common features.

When something fails or the user makes an entry error, the system clearly indicates the error with an English description of the problem, and suggestions on how to fix it.

A command line interface exists to perform any operation that can be done through the graphical user interface. Various designer utilities exist to:

Initiate promote requests. The minimum interface requires the user to enter the name of a design component or select from a list, enter the level from which to begin the promote, the target level where the promote should terminate, a flag indicating whether it's a BOM promote, and the version.

Send results from External Data Processes to a library. This utility allows the user to enter the name of a Pedigree and the target level and version to which the Pedigree information should go.

Set up and manage a private library. The utility has fields where the user can specify the name of the library (if one is to be created), the library path where the repository will reside, the userids of the owners, and either the userids or authorization groups of those who can access it. These properties can be called up for modification at any time. Whenever the owner or access fields are altered, the DCS automatically updates the authority records within the Design Control Repository as well as the operating system (i.e. AFS) permissions of the directory where the library resides.

Create and monitor a Bill of Materials. The utility offers two modes of operation. In the first, the user identifies the Bill of Materials, and enters the names of all design components to be added as members. This same utility will display any existing information for a BOM, so members can be modified or deleted. For each member, the user must indicate whether it's an input, output or support member. For an existing BOM, a function exists to revalidate all members, but this can only be done by the BOM owner. The second mode builds the BOM by reading all the information from an ASC text file written in a prescribed format. This mode can be used by designers, Data Managers, and third party tools. Regardless of how the BOM is created, a newly created BOM will result in the valid flags being set for all members. The user who creates the BOM using the first mode is automatically the owner, whereas the input file used for the second mode contains the owner information.

View process and pseudo process results. The user specifies the design component, data type, level and version. He can specify the exact process or obtain a list of all processes. For each process, the display shows the result (if it exists), the date and time it was set, how it was set (library controlled process, external process, or manually) and the criteria. These results can only be changed by the Data Manager.

Associate design problem numbers to design components. The designer uses this to pre-associate problem fix numbers to design components before they are promoted into the library. This way technical leaders and other designers can determine if a particular problem is being worked on. The interface requires the user to identify the component by name and type. Since it's not in the public library yet, it has no level or version. The

user must also supply the problem fix number. The DCS automatically assigns the "working" status to it. Later, when the designer wants to promote the component, the problem fix number will appear on the selection list, and after the promote completes, the status will change to "librared". The DCS allows the Data Manager to define a generic problem number which designers may select to associate with miscellaneous design changes that have no corresponding design problem.

WWW/Internet Access (Section 1.18)

The DCS provides a mechanism which permits access to all process and pseudo process results through the World Wide Web. Key quality control indicators can be exported out of the DCS into an accessible format by users on the WWW. Usually these results would exist in a secure repository which could only be accessed by WWW users who are working on the project. This same mechanism can be used for network access in general, including the extranets, intranets, and the internet. In addition to accessing information, the ALMs can receive special e-mail requests from users to perform these tasks:

Generate various status reports on topics such as PN-EC and Design Fix Tracking, Process & Pseudo Process Results, or BOM information. The DCS would generate the report on the fly and return it to the user's Internet or e-mail address.

If the user has the proper authority, he can submit e-mail requests to add pseudo-process information into the DCS. The contents of the mail would contain a specifically formatted command which the DCS can interpret to set the appropriate results. This could be used by people remotely connected to a project (such as the chip foundry) to send status information directly to the DCS.

The DCS permits an authorized user to send commands through the Internet Common Gateway Interface (CGI) to query information from the DCS or invoke Designer Initiated Library Processes (DILPs).

Actors & Objects (Section 1.19)

In the event of a project where a single large design team or multiple smaller ones, require their data to reside in a single repository, the potential exists for a performance bottleneck in the Automated Library Machine. The DCS offers a feature called Actors & Objects to combat this. Actors & Objects allow the Data Manager to define an alternate structure in which designers tasks are dispatched to a pool of Automated Library Machines (Actors). No design data is stored on any of them; they merely execute the tasks then store the results and data into the Design Control Repository (Object). The Data Manager can control the types of jobs each Actor is allowed to perform by creating Actor Lists. These lists contain information which the DCS uses to determine which ALM to route a particular job to. FIG. 9 shows an Actor/Object environment with four Actors. Jobs involving the data type of layout and timing are segregated to ALM4. All remaining work is sent to ALMs 1 through 3. The DCS determines which to use based on an mechanism which tries to find either a free ALM or choose one that may be able to spawn a parallel process (assuming the operating system supports it).

Importing and Tracking Data (Section 1.20)

Internally the DCS tracks all data by component name, data type, level, version, library and most importantly a file reference (fileref) number. These six attributes give every piece of data in the system a unique identity. In a private library, all data is tagged with a DCS identifier as part of the

filename, but the identifier may or may not be unique. This is because private libraries don't have a concept of levels, versions or file references. They are merely working areas for the designer, and only require the data to be identified by name and type. The system permits the designers to have multiple copies of a design component by using iteration numbers to distinguish between recent and older data. However, even though the concepts don't apply, the DCS still assembles an identifier and tags the data. There are two methods by which a piece of data can appear into a private library.

1. The designer creates the data from within the private library using some tool (Schematic editor, text editor, circuit simulator).
2. The data is created by some tool completely outside of the private library, but the designer wishes to import it into the library.

In either case, the tool (or user) chooses the filename. By default, this is the design component name. In the first case, the designer will be asked to specify the data type either prior to, or during invocation of the tool. In the second case, the user will be prompted for the data type during the import. In both cases of a data type entry requirement the DCS will automatically default the version, level and file reference number in order to assemble a uniform identifier code. This code will be appended to the design component name and will become the new name of the object. Upon promotion from a private library into a public library, the DCS will automatically assign a real file reference number to the object. Based on the destination version, and level, the DCS will assemble a new identifier and rename the object accordingly. The file reference number remains the same for the life of the object. As the object traverses through the levels of the library, the level is the only piece of the identifier that changes. In addition, the DCS maintains the same identifier information internally. This is considered the official tracking information and is always updated first during a promotion or installation of a new object into a public library. The object renaming is done afterwards. Appending the identifier to the object name serves two purposes:

It increases data security by providing a way for the DCS to check data integrity during promotions. The information contained internally must match the external identifier at the start of a promote. A mismatch signifies possible tampering of the data outside of the DCS, and the Data Manager is alerted to the mismatch.

It provides an alternate way for a user or another tool (such as the library search mechanism) to ascertain the version, level, and data type of an object simply by looking at it. This contributes to the availability by providing a means to locate and access data even if the Design Control Repository is unavailable (i.e. server down).

One major advantage to this tracking scheme is it's independent of the physical location of the data. The DCS permits the Data Manager to establish as many repositories as he needs down to any level of granularity. For example, all data for a library could reside in one physical directory, the data could be segregated by version only, or there could be separate directories for each type of data. This level of flexibility allows the Data Manager to optimize the library to a given environment. For example, he can define his repositories in such a way that the data which moves most often is located on a single volume on his fastest server. Data which never moves (i.e. Release Level data) can be located on slow servers or spread out over multiple servers. As the Data Manager defines his library structure, he can specify

the locations for every level of each version. In addition, if he has specific data types that he wishes to further segregate, he can specify a location for them. Finally, the DCS supports a feature called Automatic Component Grouping in which all data types for a given component name will automatically be located in a subdirectory off of the level directory. FIG. 10 illustrates a portion of a library directory structure with different levels of storage granularity. LIB_DIR is the primary directory for all data in the library. Under it, data is segregated by version where version 1 data resides in the subdirectory VERS1. At this point the diagram illustrates three examples of further segregation. In the VERS1 directory are the schematics and behaviors which comprise level E1 and E2 for all 3 design components. Although they are physically mixed together, their unique identifiers allow the DCS and users to tell them apart. The diagram shows the circuit layouts to be further segregated by data type. So they reside in subdirectory TYPE_LAYOUT. Once data reaches level E3, it is segregated by level and type. LEV_E3 contains all the schematics and behaviors for the E3 level, but the layouts reside in the TYPE_LAYOUT directory under LEV_E3. The final example shows data segregated only by level with no regard to type. This is seen in the release level repository LEV_R1. By offering this kind of flexibility, the DCS permits the Data Manager to group the data in the most advantageous way. In addition, the Data Manager could invoke Automatic Component Grouping, which would result in further subdirectories under VERS1, LEV_E3 and LEV_R1 to segregate the pieces by component name.

Note: This is unnecessary in the TYPE_LAYOUT directories since the only difference between the objects is the component name. In order to boost performance, every time a structural change is made to a library which involves repositories, the DCS automatically generates a master cross reference between library/level/version/type and physical location. This table is used by mechanisms such as the library search engine to locate data without requiring extensive querying of the Design Control Repository. It also enables library searches to occur in the event the Design Control Repository is unavailable.

Preferred Embodiment for Managing Shared Libraries (2.0)

The present embodiment provides a controlled environment for the acquisition, movement, disposition and removal of data from a Data Management System. The embodiment is described from the perspective of an overall algorithm which manages data Libraries. This encompasses not only the storage management issues but the necessary interaction with a centralized Data Control Repository.

Our embodiment covers a broad array of implementations ranging from a system whereby the user constantly interacts with the Data Control Repository to acquire ownership, deposit or move up through the system described in the preferred embodiment which incorporates Automated Library Machines (ALM) with a sophisticated file movement algorithm.

The Library Management algorithm serves as the interface between the user and the Data Management system for routine data control functions. Our preferred embodiment interacts with the Lock and Authority Manager to permit an environment which allows multiple users to possess ownership in a data object, but ensures only one owner is updating an individual instance at any given time. A Check Out utility is provided for the user to request ownership to a piece of data, transfer ownership, or take ownership if they are an authorized surrogate of the current owner. Likewise,

a utility exists to perform data deletion in a safe and controlled manner by ensuring only authorized Data Managers or valid data owners delete their own data without jeopardizing any other data.

An integral part of Library Management is establishment of private and public libraries. Often these public libraries are shared by many users which can create data integrity exposures if data is not properly stored into and moved through a public library. The overall algorithm manages the movement of the data between the actual physical locations specified under the established library structure. Since our embodiment permits data to reside across different computer platforms, the algorithm contains functions for handling cross-platform data transfers. Although our Library Management algorithm only requires a simple promotion algorithm in order to function, the present embodiment reveals a highly sophisticated File Movement Algorithm.

Lightly-loaded Data Management Systems can usually support execution of the above Library Management functions in the client's environment where the user calls upon the Data Control Repository to initiate the function, and the repository immediately invokes the appropriate algorithm or utility. However, in large enterprises, this can lead to unacceptable performance degradation as many users simultaneously access the repository. Therefore, our Library Management algorithm is capable of supporting Automated Library Machines arranged in a variety of configurations to optimize performance. The use of ALMs also permit Automated Library Processing to occur. The Library Manager incorporates routines for properly installing any output created by an Automated Library Process into the DMS.

Finally, the Library Manager provides instant notification to the user for any service rendered. This occurs whether the task is executed in the user's environment or remotely on an Automated Library Machine. In the event a task fails to complete, error messages explain the problem. Successful operations also result in notification thus ensuring users possess situational awareness of their data at all times.

Our preferred embodiment describes a File Movement Algorithm which interfaces with other Managers in the overall Data Management System. This provides a great degree of data security while offering a plethora of automated features. The Promotion Algorithm interacts with the Authority and Lock Managers to ensure that users transfer data that they own from a private library into a public shared library. Furthermore, only authorized users may promote the data through the various library levels.

Upon initiating a promote request, the algorithm compares any recorded process results against pre-defined promotion criteria to ensure that only data which meets a quality standard may be elevated to the next level. The promotion algorithm offers a flexible means of promoting large volumes of data including features which handle heterogeneous tapes of data from different levels and versions within the same request. Interaction with the Aggregation Manager allows fast and efficient Bill of Material (BOM) promotes.

The promotion algorithm also works in conjunction with the Problem Fix and Release Manager to apply problem fix tracking, incremental change, part number and release control to any desired piece of data moving through the Data Management System. This includes interaction with the user to gather the appropriate information at promotion time as well as background checking to safeguard against data integrity violations such as a single part being associated with two different releases.

For environments such as preferred embodiment, which incorporate Automated Library Machines, our promotion

algorithm permits the user to precheck a work request interactively prior to the request being sent to the ALM. This feature ensures the work request will complete successfully by running all the same checks which are normally run by the ALM prior to data movement. Since our embodiment permits the user to request a promote through multiple levels with a single invocation, this pre-qualification feature can greatly improve productivity.

In order to maximize throughput of large data volumes, our Library Management Algorithm employs Automated Library Machines to act as independent agents for the user community. These ALMs are service machines which use a virtual queue to accept work requests from users and perform Library or non-Library functions. The ALMs interface directly with the Data Control Repository through dedicated high speed ports in the Communication Manager. They can exist on clients, servers and on different computer platforms.

Our preferred embodiment describes three basic configurations which permit the ALMs to perform any of the services requested by the Library Manager algorithm. The basic configuration is known as a Conventional system where a single ALM accepts all work requests and handles all services for the Library Manager, including any Automated Library Processing. The second configuration is Remote Execution Machines which is an extension of the Conventional system. Here, a single ALM receives all work requests from the user, and processes all promotion, installation, movement, and removal of data. However, additional ALMs may exist to perform Automated Library Processing. The ALMs interact with the Library Manager, Communication Manager and Promotion Algorithm to dispatch any desired library processing to a Remote Execution Machine, which executes the task and returns the results to the master ALM. The most powerful configuration is known as Actor/Objects and this arrangement employs a pool of ALMs which serve as general purpose machines. They can perform any desired Library Management function, including Automated Library Processing. They can even interface with Remote Execution Machines to provide an environment with both general purpose machines and dedicated service machines. Each ALM can be programmed by the Data Manager to define the type of work requests it can process. This arrangement even includes a special Dispatcher ALM whose sole purpose is to dispatch user work requests to the next available Actor machine.

Automated Library Machines are special purpose Auto-Reader service machines which means they are capable of performing virtually any software task that can be invoked from a command line. These tasks can be completely independent of the Data Management System which permits an ALM to play multiple roles in a business environment. It can literally be processing a Data Management request one minute, and formatting a word processing document for printing the next minute. The underlying AutoReader mechanism incorporates features to enable automatic recovery in the event of a system crash. The Library Manager further enhances this by adding automatic retry of aborted library operations due to system problems.

File Promotion Process

The present embodiment incorporates a robust process for promoting data from a private library into a shared public library as well as moving data through a shared public library. Although it's especially suited to interact with Automated Library Machines and the other algorithms described in the other sections of the Preferred embodiment, this process does not require any of those elements to be present in the Data Management System (DMS).

In order to track data properly, our embodiment provides a conduit for data to enter, and travel through, the DMS in a safe and controlled manner. This ensures that all data is subjected to the proper checks regardless of the point of origin or final destination. The preferred embodiment depicts the overall flow of the promote, or data transfer, in FIG. 12.

The flow begins with Step 22101 in FIG. 12 in which the user is presented with the Promotion Screen. FIG. 13 shows this screen, which permits the user to enter information about the file(s) they wish to process. Promotion can entail transferring data from the user's private library to a public library, or moving data within a public library.

Turning our attention to FIG. 13 we see the promotion screen which consists of data entry fields 22201 through 22208. A single screen is used to either Put a file from a private library into the DMS or Promote a file from one level of the DMS to another. The type of action is determined by the user supplied information.

The preferred embodiment presents the user screen in a graphical environment where the user engages pull down menus, pop-up menus, drop-down lists, radio buttons, push buttons, fill-in fields, and mouse interaction. It should be noted, however, that all functions discussed further in the preferred embodiment can be implemented using simple text screens, or more advanced data entry systems such as touch screens, voice commands or 3-D graphics. The preferred embodiment depicts the method most conducive to the Motif™, Windows™, and OS/2™ application environments.

Field 22201 holds the name of the file to be promoted. If a single file is being promoted, the name is typed in directly. If the user desires a selection list, it is automatically invoked by simply leaving the field blank and completing the remainder of the form. Upon hitting Enter, a library search would be employed to obtain a selection list of files. The third method is to promote a group of files via a text-based list, edited in advance, in a prescribed format. This is covered below in the explanation of user options.

Fields 22202 and 22203 allow the user to enter the Version and Library File Type respectively. In the preferred embodiment a promotion is not permitted across versions nor can a file type change as a result of a promote.

Fields 22204 and 22205 convey the Source Library and Level information. In the case of a Put, the Source Library can be any valid private library in the DMS. The Level would default to User, but a valid entry level, name can be entered. Field 22204 can also be the name of a public library. In this case, field 22205 would contain the starting level for the Promote. Each field has a "smart" drop down menu button associated with it. The Library button next to field 22204 displays a list of all public libraries in the DMS plus all private libraries owned by the user. The button next to field 22205 displays all valid levels for the Library entered in field 22204.

Fields 22206 through 22208 are almost identical to fields 22204 thru 22205, represent the Destination information. Field 22206 is the destination library which is frequently the name of a public library. As with field 22204, the drop down menu button next to field 22206 displays all libraries in the DMS. However, our embodiment also permits a private library owned by the user to be a valid entry. In this case, fields 22207 and 22208 default to User, but can be changed to the valid level names, and none of the options at the bottom of the screen apply. The resulting operation would be a simple file copy from the source private library to the destination private library, with the file being named according to the Destination Level information.

Returning to the general case, if field 22206 contains a public library then the user would fill in the destination level in field 22207. Once filled in, field 22208 automatically takes on the same value. In the case of a simple Put, field 22207 represents the final destination and is equivalent to the Destination Entry Level in field 22208. For a Put with Promote, field 22207 represents the destination level while field 22208 denotes the doorway through which the file should enter the DMS. Finally, in the case of a Promote, field 22207 again represents the Destination Level and field 22208 is ignored.

Turning our attention to the lower portion of the screen, we see two sets of push buttons. The first set, 22209, represent user options that pertain to any type of Put or Promote. Any number of these options can be invoked and they are defined as follows:

Via List The list of files to be processed will be read from a text based file. The user is presented with a dialog box requesting the name of the file. The names of the files will be read from this list, but all other information will be taken from fields 22202 through 22210.

Foreground Checking Pre-checks the data using the same checks that will take place during the background portion of the promotion. This allows the user to test the promote and ensure everything will work before the request is submitted to the DMS.

Via Copy is normally active for most environments. It enables the DMS to use a file copy operation as the preferred method for transferring data from the source to the destination. In certain environments, this may not be possible or desirable, so deselecting this option forces the data to be "sent" from the source to the destination.

Reset Update Lock is the means by which the user informs the DMS to remove any existing Update Lock from the file(s) and leave them in an unowned state at the completion of the promote.

High Priority Tells the DMS to process this request next assuming it's the only high priority item in the queue. If there are multiple requests with this priority, they are processed in FIFO order.

Override Process Params Allows the user to add or modify the parameter passed to an underlying library process.

The last set of buttons is set 22210 which represent options only available during a Promote. This is because these options only pertain to data already under control of a DMS. Any number of them can be selected and they are:

BOM Promote Indicates the user wants the Bill of Materials associated with the file entered in field 22201 to be promoted. If the file doesn't have a BOM associated with it, the promote will fail.

Anchor Only Indicates to the DMS to only promote the anchor file of the BOM associated with the file entered in field 22201, but leave the members at their current locations.

Retain Source is usually "off" which enables the DMS to move the file from the source location to the destination. However, in certain environments it may be desirable to leave a copy of the file at the source location after the promote is completed, so this option exists for this purpose.

Returning to the overall flowchart in FIG. 12, information entered in Step 22101 is now passed to Step 22102, Foreground Processing. The detailed algorithm for processing promotion requests in the foreground is described in FIGS. 14a thru 14f. It begins with Step 22311 of FIG. 14a, Parse

Opts. Here all the information entered in Step 22101 is checked for validity. The following cases are examined:

If the Source and Destination Libraries are private libraries, then the Source and Destination Levels are allowed to be any combination of User and valid public levels. The user must have edit authority to the Destination private library. This results in a simple file copy with the name being adjusted according to the Destination level field.

If the Source Library is a private library and the Destination Library is public, then the following conditions must be met:

1. If the Source Level is User then the Destination Entry Level must be a valid entry point. The Destination Level must be the same or higher.
2. If the Source Level is a valid entry point, then the Destination Entry Level must be identical. The Destination Level can be the same or higher.

If the Destination and Destination Entry Levels are the same, this classifies as a simple put. If the Destination Level is higher, it's a combined Put with Promote.

If the Source Library is a valid public library, and the level is a valid level, then the Destination Library must be the same as the Source Library and the level must be higher than the Source Level. These conditions signify a Promote. In this case the Destination Entry Level is ignored.

Any other combinations of those fields is considered an error condition and the promote terminates. In addition, the options are examined to set various flags which will be used to determine branching later in the algorithm.

Step 22312, Input Files follows Step 22311. Here, FIG. 13, entry field 22371, Name is examined for three possible responses. The simplest case is the name of a single file which implies only this file is to be promoted. Second, is a blank or some other keyword in which the user requests a library search of all files of the specified Library File Type starting at the specified Source Library, Level and Version. If any of these fields are invalid or missing, the user is prompted to enter the information. The standard Library Search manager is invoked to perform this task. At the completion of the search, the user is presented with a selection list of all files found. One or more files can be selected for promotion. The third possibility is that the ViaList user option in button field 22379 of FIG. 13 has been selected. This indicates that the user wants the foreground process to extract the names of the files from a text file. The user is prompted for the name of the file which lists the names of the files to process. All other information, besides the file name, is taken from the main screen.

Once the input file(s) are determined, the algorithm enters the File Loop in Step 22314. For the simple case of a single file being promoted, this loop is only exercised once. In the cases where a selection list or text file was used to provide a list of data to promote, all Steps between 22313 and 22347 are executed for each file.

Step 22315 asks the question "is this a Put or Promote?" The answer to this question determines which way the program must branch. A Put refers to data entering a public library from a private library whereas a Promote refers to data moving through a public library. Our embodiment permits a Put followed by a Promote in the same request, but for purposes of resolving Step 22315, that request is treated like a simple Put. All Promotes continue to Step 22326 in FIG. 14c while Puts branch to Step 22316, Fig Info in FIG. 14b.

In Step 22316 of FIG. 14b, the program requests any Automatic File Group relative to the file being promoted.

This information includes a flag indicating whether the candidate file is the master file of a file group, and if so, the list of Library File Types associated with that file group. By definition, all members of a file group share the same file name, library, version and level.

Step 22317 reviews the information returned in Step 22316 to see if a File Group Exists. If so, the program continues with Step 22318. Otherwise, it branches to Step 22321. Steps 22318-22320 pertain to processing file groups.

In Step 22318 the program checks for the existence of any Required Members of the File Group. These are denoted in the information returned in Step 22316. If a required member can't be found in the user's private library, an error message is issued and the promote is terminated. If an optional member doesn't exist, a warning is issued to the user, but the promotion continues.

Step 22319 examines the Check Option whose flag was set in Step 22311. If it's true, then the locks on the all existing subordinate files are examined, in Step 22320, Fig Locks, to ensure the user either owns the Update lock or is a surrogate for the owner. In addition, it ensures no other type of lock exists which would prevent the Put. Such an example would be an Overlay lock at the entry level. If the user is not the owner, our Lock Manager will Update Locks.

In our preferred embodiment, our embodiment maintains absolute data integrity by adhering to the following rule. The Foreground Check option, available to the user on the promotion screen in FIG. 13 only pertains to the foreground processing of the promote. It is ignored in the background, where full checking is done at all times. The option serves merely as a performance enhancement which allows the user to submit promotion requests with minimal checking. The advantage of the Check option is that an error free foreground session practically guarantees a successful promote in the background. However, depending on the environment and the implementation, the trade-off may be lengthy processing time on the client machine. For these situations, our embodiment offers the user the option of bypassing a series of foreground checks and "taking their chances" on the promote processing successfully in the background.

Steps 22321 and 22323 can be processed in either order, and they are used as decision points for Steps 22322 and 22324 respectively. In Step 22321, the program inspects the Fix Management information for the package. If the package is under Single Fix Mode or Engineering Change Mode, then the LFT FM, or Library File Type Fix Management, flag is examined for the LFT being processed. If the flag is on, Step 22322 is invoked, otherwise the program proceeds to Step 22323. In Step 22322, FM Assoc, the program attempts to associate the file being processed with a Problem Fix Number. If the package is under Single Fix Mode, the default Problem Fix Number is associated to the file. If the package is in Engineering Change Mode and the repository already has existing Fix Numbers for that file, they are presented on a user screen. This is usually done with another interrogation of the Control Repository. The user is given the choice of selecting one of the existing numbers or entering a new one. If no Fix Numbers exist, the user is prompted to enter a new one.

In Step 22323, the program checks to see if the Part Number Control Level is being crossed by comparing it to the Destination Level provided by the user in Step 22301. If the answer is "yes", then the LFT PN, or Library File Type Part Number, flag is examined for each LFT being processed. If the flag is on, Step 22324 is invoked, otherwise the program proceeds to Step 22325. In Step 22324, PN Assoc, the program attempts to associate the file being processed

with a Part Number. If the repository already has an existing Part Number for that file, it is presented on a user screen. This is usually done with another interrogation of the Control Repository. The user is given the choice of selecting the existing PN or entering a new one. If no Part Number exists, the user is prompted to select a new one.

Since Steps 22321 through 22324 may be repeated for a large number of files, the Design Control System may be implemented in such a way that the DMS server returns all associated Part Number and Fix Management data for all files being processed with one large query. The client then sifts through the data to find the information necessary to interact with the user. The algorithm states the information that must be obtained from the Control Repository and the user, but permits a great deal of flexibility in the way it is acquired.

At this point the algorithm continues with Step 22325, Dest=Entry in FIG. 14d. In this step, the code determines if this is a simple Put or a Put/Promote combination by comparing the Destination Entry Level with the Destination Level. If they are the same, it's considered a simple Put. In this case the code proceeds to Step 22319. If the Target Level is higher than the Entry Level, it's considered a Put with Promote and continues to Step 22332.

Returning to the Put/Prom decision in Step 22315, if the user is requesting a promote, the code branches to Step 22326 in FIG. 14c. Here the control repository is queried to find out whether the file being promoted is the anchor file to a Bill of Materials (BOM). Regardless of the outcome, the Model Option flag in Step 22327 is examined. The answer to both questions yield four possibilities. If both answers are "yes" or both answers are "no" this is the normal case in which the algorithm proceeds to Step 22330. If the file is a BOM, but the BOM Promote Option was not specified in Step 22301, the Anchor Only Option flag in Step 22328 is examined. If this flag is on, then the user is requesting to promote only the anchor file and not the entire BOM. If the flag is off, a warning is presented to the user requesting that either the BOM Promote or Anchor Only Option must be selected. Finally, if the file is not a BOM, but the user specifies the BOM Promote Option, an error message is displayed and the program terminates.

Continuing with Step 22330, the program checks to see if a BOM exists at any of the levels being promoted through with the same name as the file. If so, a warning is issued, in Step 22329, to the user indicating a BOM will be and subsequently deleted if the promote continues. The user is given the option to continue or abort the operation. If no BOM Overlay is imminent, or the user accepts the overlay, the algorithm proceeds to Step 22332 of FIG. 14d.

In Step 22332, Chk Lvl Order, the promotion path is examined to ensure a legal path exists from the Source Level to the Destination Level. In the case of a Put, where the Destination Level is not equal to the Entry Level in Step 22325, the Entry Level is also checked to ensure it's part of the path. If not, an error is displayed and the program terminates. Step 22333 is a Level Loop which must be set up so Steps 22334 and 22335 can be performed for each level in the promotion path.

Step 22334, EC LVL, is where the program checks to see if the EC Level is being crossed during the transport. This is done by determining if the Destination Level entered in step 22301 is at or above the EC Control Level. If the test resolves to a "yes" answer, the program must perform step 22335, EC Assoc. Here it queries the Control Repository for EC information regarding any Problem Fix Numbers and Part Numbers selected or entered in steps 22322 and 22324.

For any Fix Number or Part Number not associated with an EC, the program prompts the user to select an EC from a list of existing EC Numbers or choose a new one.

Upon completing the loop, the Check Option flag in Step 22319 is examined again. If this flag is off, the program proceeds to Step 22345 in FIG. 14f. Otherwise, if the user requests the option, the Level Loop in Step 22333 is again invoked. Step 22337, Lock Check is performed to ensure no locks exist against the file at any of the levels being traversed. These include Update, Move, Overlay and Processing locks. If Update locks exist, the user must either be an owner or a surrogate of the owner. In the latter case, the user is given the opportunity to reset the lock, upon which notification would be sent to the current owner.

In Step 22338, the User's Authority is checked to ensure they can promote the file to that level. Finally, in Step 22339, a complete BOM Invalidation check is performed. There are two forms of this check. In the simplest case, the file is not a BOM, so the control repository is examined to see if any other BOMs will be invalidated by the movement of this file to the Destination Level. In the more complex case, the file is a BOM. Here the same check is made as in the simple case, but additional checks must be made for every member of the BOM to determine if their movement will invalidate any other BOM. In either case, the results of the pending invalidation are displayed for the user to examine, using our Aggregate Manager for this check, as BOMs may have many members.

It should be noted that the order of Steps 22337 through 22339 is not critical and is combined into single large queries in our preferred embodiment. Files being promoted from a Private Library into a Public Library employ the QRSUPGET function, while promotions within the Public Library utilize the QRSUPCHK routine, described in FIG. 16. Upon completion of these checks for each level, the program moves on to Step 22340 to ensure the File Exists. This not only means the file physically exists with the proper nomenclature in the directory corresponding to the Source Level, Package, Version and Library File Type, but this also means the control repository agrees that the file exists in that location. If for example, the user believes the file is at Level A, but the control repository is tracking it at Level B, this is a data integrity violation and the promote ceases.

At this point, the algorithm proceeds to Step 22341 in FIG. 14e, where the question "is it a simple Promote" is asked. If so, Step 22342, Criclk is performed. For a promotion beginning at a library level (not one that is chained to a Put), the control repository is queried for any Post-Processes or DILPs defined at the Source Level. If there are any processes, then the process results for the file being promoted are examined to ensure they meet the proper promotion criteria. If any fail criteria, a warning is issued informing the user that the promotion will fail.

Next, Step 22326 is executed again to check if the file is a BOM. If so, a Member Loop is set up in Step 22343 where each member has their process results checked against any Post-Processes or DILPs which exist for those LFTs at the Source Level. This checking is the same as Step 22342.

If the answer to the Promote question in Step 22341 or the BOM test in Step 22326 is "no" the code proceeds to Step 22345. Likewise if the Member Loop in Step 22343 was exercised, the code proceeds upon exit to Step 22345 in FIG. 14f. In Step 22345, Fproc Option flag is examined to see if the user is bypassing Foreground Processing. If the flag is off, the code returns to the top of the File Loop in Step 22314. If the flag is on, the Level Loop in Step 22333 is initiated again. For each level that the file will pass thru, Step 22346, Fproc will be called.

In the Fproc step the code queries the control repository for all foreground processes defined for that Library File Type at that level, version and package. Each process is then executed immediately in sequence so the user can enter the appropriate responses. These foreground processes are established by the Data Manager. Upon completion of the Level Loop, the code returns to the File Loop in Step 22314 of FIG. 14a. Once Steps 22314 thru 22346 are executed for all files in the promote request, the algorithm proceeds to Step 22347, Override Option.

In Step 22347, the Override Option flag is examined. If it's off, the program proceeds to Step 22349. If it's on, then Step 22348, Override is executed. In this step the user is allowed to override any of the process parameters for the library processes that will be executed during the promotion. The algorithm queries the control repository for all Pre and Post-Processes defined for this Library File Type at this level, version and package. All of the process parameters are displayed on the screen, and the user can select and modify as many as desired. Once finished, the user "OK"s the modifications, and they are written into the control information which is used in the next step. The DMS will use these modified parameters to drive the library process, only if they exist. Otherwise it defaults to using those define by the Data Manager.

In step 22349, Xmit, the program gathers and transmits all of the necessary data to the Design Control System. In our preferred embodiment, the destination would be an Automated Library Machine which would access most of the information via a "copy" operation. The following types of information need to be transmitted:

The type of request: Put, Promote, or Mixed

The list of files being promoted. The following information must exist for each file in the list:

The Filename

The Library File Type

The Package

The Version

The Source Level, Entry level, Destination Level

Fix Numbers (if any)

Part Numbers (if any)

EC number (if any)

Any user selected options that pertain to the background operation.

The user's electronic id or e-mail address.

Information gathered during any foreground processing that occurred.

Information relating to any process parameters altered during the Override operation.

In addition, for Put operations the files themselves must be transferred from the user's private library. This includes all existing members of Automatic File Groups. Our preferred embodiment performs this by either having the Automated Library Machine copy the files or by sending them. The determination is made based on a user option called Via Copy which exists on the main input menu in FIG. 13. One other option on the main menu is the Emergency option which informs the DMS to treat this as the highest priority in the processing queue. This feature enables critical work to be processed ahead of older jobs.

Returning to FIG. 12, the algorithm proceeds with the Background processing in Step 22103. In our preferred embodiment, the promote request from Step 22102, Foreground is transmitted to an Automated Library Machine (ALM) for processing. The first step is for the ALM to Read Control Information which is done in Step 22411 of FIG.

15a. Here, the names of files in the promote request are stored into a data structure along with all pertinent control information like Problem Fix numbers, Part Numbers, EC Numbers, Entry Level & Destination Level for each file, and the user options. The options are further parsed to set flags. In our preferred embodiment, these requests are homogeneous which means all files in the request move from the same source to the same destination. Since the Foreground step determined the type of request, it's contained in the control information and used to set the flag which is examined in Step 22413.

However, our embodiment also supports heterogeneous file movement. An example of this is two files where one moves from level A to B while the other moves from level C to D. Although this request can't be generated by the Foreground process in Step 22102, it can be created by 3rd party tools interfacing with the DMS. To accommodate this, the control file indicates the type of request as Mixed. In this case, Step 22413 must employ the same algorithm used in the Input Files step of FIG. 14a to determine if the request is a Put or Promote on a file-by-file basis. Since the control information contains the Source, Destination and Entry Level for each file, this is quite easily done. At this point, the File Loop in Step 22412 is entered in order to execute Steps 22414 thru 22418 on each file in the request. Step 22413 checks to see whether the request is a Put, Promote, or Mixed where:

Put means all files in the request are being promoted from a private library into the same level of a shared public library. The files may have to be promoted one or more times to reach the destination, but all files will travel the same path.

Promote means all files in the request are being promoted from one level of a shared public library to another. The files may have to be promoted one or more times to reach the destination, but all files will travel the same path.

Mixed means the request contains a mixture of files where 2 or more files may travel different paths. In this case the background algorithm must determine the promotion path for each individual file.

Note: Our preferred embodiment does not allow this type of request to be generated from the Foreground user interface, but 3rd party tools may create one.

In the case of a Promote (ie. the file is being moved from one level of the DMS to a higher level), Step 22414 is employed to set an Overlay Lock on the file. The DMS uses this Overlay Lock to prevent a different ALM from moving the same file while this request is in progress. Next, Step 22415, Promote Check is called upon to do the following checks:

Ensure that the Source Level is valid and not a frozen Release Level.

Ensure that a valid path exists from Source Level to the Destination Level.

Determine the next higher level above the Source Level. (This may not be the Destination Level if this is a multi-level promote).

Obtain the physical location of the Source and Destination Levels.

Ensure the user is authorized to do this promote. For regular promotes, the user must have normal promote authority whereas for BOM promotes, Model Promote authority is required.

Ensure the file really exists in the DMS at the expected Source Level.

Check for any Processing or Move locks. Also look for any Overlay Locks at the next level.

Check all Post-Process and DILP criteria at the Source Level to ensure the file meets all of the criteria. In the case of a Bill of Materials (BOM) promote, each member of the BOM is also checked to ensure all criteria is met.

If the file is crossing the EC Control Level, and the file is under Problem Fix Management, check to ensure a proper EC number is provided.

If a BOM Promote is requested, a check is made to ensure a BOM is associated with the file being processed.

If a processing lock is detected, the promote request is recycled back into the DMS queue to await completion of the library process.

Note: If a processing lock exists, special hang detect code is used to determine if the file has been locked for an unusually long period of time. If so, the user is notified and advised to check on the process job.

Returning to Step 22413, if the request is a Put, then Step 22416 in FIG. 15 is employed to see if Update Locks Exist in the user's name, at the Destination Entry Level, for the file being processed. If not, one is acquired in Step 22417, Updt Lock. This lock may be permanent or temporary depending on the setting of a user controlled option which is discussed later. The reason for setting it at this time is to prevent someone from taking ownership of the file while the Put is in progress.

In Step 22418, Put Checks, a series of checks are performed against the file. Many of these are duplicates of those run in the Foreground in Step 22102, if the user specified the Check option during Step 22101. In our preferred embodiment, these checks are all done with a single query to the Control Repository. The following are performed:

A check is made to ensure no Processing, Move or Overlay locks exist on the file.

Ensures the user is authorized to Put this file to the Entry Level.

The physical location of the Entry Level and Source Level are acquired.

The File Reference number is generated.

The Lock Reference number is acquired.

The Entry Level is checked to ensure it's a valid entry point into the DMS. It may be a Sideways Release Level, but not a regular Release Level.

If any of the checks fail, the promotion terminates with an appropriate error message.

At this point the File Loop in Step 22412 is repeated until all files are exhausted. Upon exit from the loop, control proceeds to Step 22419, List Files. Here, the file information is re-written with the source and destination physical locations in preparation for the upcoming file transfer. The file name of this control file indicates whether the file transfer pertains to a Put or Promote.

Upon completion of List Files, the Pre-Processing Done flag is examined in Step 22420 of FIG. 15c. The flag is set to "Done" if no Pre-Processing is required, or whenever the Process Manager completes all required PreProcesses. If the answer is "no", then the Process Manager, in Step 22421, is employed to run any pre-processes that exist. The Process Manager processes all files in the list with a single invocation using the control file generated in Step 22419. If any of the pre-processing fails, the promotion with an appropriate error message. At the completion of the Pre-Processing, the PreProcessing flag is set, and the Process Queue in Step 22422 is interrogated. Although the processing completed in

Step 22421, it may have required work to be dispatched to other ALMs. In this situation, the DMS maintains data integrity by setting a Processing lock in the Process Queue. If the current ALM detects an outstanding lock in Step 22422, on the file being processed, it will recycle the promote request until the lock is cleared. This ensures that the order of execution for library processes is always maintained regardless of the distribution of workload.

Continuing with Step 22423, a special option flag denoted Nogo is checked. If it's set, the promotion terminates successfully even though no files were actually transferred. This is a mechanism the DMS uses to initiate library processing against a transient file which doesn't need to be permanently retained. For example, a transaction file may be promoted into the DMS for the sole purpose of initiating a library process to update a master file residing in the library. Once the master file is updated, the transaction file is no longer necessary so it can be discarded. In this case, the Nogo option terminates the Put without wasting time transferring and deleting the file. If the Nogo option doesn't exist, control proceeds with another File Loop.

Once again, Step 22412 is invoked to loop through all files in the request. In addition, Step 22413 is used to determine if this is a Put or Promote. In the case of a Put, Step 22424, Sup Put is exercised. In our preferred embodiment, this is performed via the QRSUPPUT routine, which is responsible for updating all the necessary tables to indicate that the file now resides in the new Entry Level location. In addition, the Control Repository examines the Fix Management and Part Number flags for the current Package, File Type, Version, and Level (PFVL). It also checks to see if the EC or Part Number Control Levels are being crossed. Depending on the results of these flags and levels, it expects the corresponding Problem Fix numbers, Part Number and/or EC Number to be available. This information was gathered during Step 22102, Foreground Process. The DMS updates all tables associated with Problem Fix Management and Part Number control at this time. If the file is overlaying an older version of the same file, the Problem Fix numbers from the old file are appended to those of the new file. Any previous iteration of the file, at a higher level, with the same Problem Fix Numbers as the file being promoted will result in the higher level Problem Fix Numbers being Superseded. Also, the locations of the File Group subordinate files are updated. If this query completes successfully, the file is officially promoted into the DMS, even though the file hasn't physically been transferred yet. In order to maintain absolute data integrity, the DMS always has the correct information. If a file physically resides in a state other than that reported by the DMS using a query, the file is in error and needs to be rectified.

If Step 22413 determines the request is a Promote, Step 22425, Sup Prom is invoked. In our preferred embodiment, this is accomplished by the QRSUPPRM routine, which is responsible for updating all the necessary tables to indicate that the file and all File Group subordinate files now reside at the level above the Source Level. For BOM promotes, the tables for all member files previously at the Source Level are updated as well. Members currently at the Destination Level or higher, or members in other libraries, will not be affected. In addition, the Control Repository examines the Part Number flag for the current Package, Version, Level and LFT. If it exists, and the Part Number Control Level is being crossed, it expects the corresponding Part Number to be available. Likewise, the DMS checks to see if the EC Level is being crossed, and if so, the appropriate EC Number must be present. This information was gathered during Step

22102, Foreground Process The DMS updates all tables associated with Problem Fix Management by appending any new Problem Fix Numbers to any existing numbers pertaining to the previous iteration of the file. Any previous iteration at a higher level with the same Problem Fix Numbers as the file being promoted, will result in the higher level Problem Fix Numbers being Superseded Subsequently, the DMS attaches all Problem Fix Numbers for this file to the EC Number. Also, the Part Number tables are updated, as necessary.

At this point control proceeds to FIG. 15d where various flags must be checked, and possible additional actions taken. In Step 22426, BOM Ovly, the Sup Put or Sup Prom return code is checked for an indicator that an existing BOM was overlaid by this promotion. If so, then Step 22427, Send Msg is invoked to send a message to the owner of the BOM informing them of the BOM eradication. Next, Step 22428, Erase Toll-Fig is invoked to check if a file group exists for the file being promoted. If so, and the Erase-To-Level flag is set, then Step 22429, Erase Fig is invoked. Here, the program determines whether the subordinate file will be replaced by an incoming file. If not, the existing subordinate file is erased and a message is sent to the owner informing them of the file removal.

In Step 22430, the BOM Invalidate flag is checked. This indicates that the promotion of this file caused one or more BOMs somewhere in the DMS to be invalidated. This would happen if this file is a member of some other BOM. A sophisticated algorithm exists in the DMS to quickly check all BOMs in the system for the presence of this file. If an invalidation occurs, Step 22431 is invoked to Notify BOM Owners exactly which BOM was invalidated and which file caused the invalidation. It should be noted that Steps 22426 and 22428 and 22430 can be checked in any order.

The algorithm continues with Step 22432, Move Files. The method for moving data is very dependent upon several factors. These include the system environment, the existence and/or arrangement of Automated Library Machines, and the user options selected during initiation of the Put or Promote. If no ALMs exist in the DMS, it's assumed that the user's client environment has the proper read and write access to the data repositories. The code uses the appropriate combination of copy, delete and/or rename commands to accomplish the desired Move operation. However, if ALMs are employed, a Move algorithm is used to determine how this step is implemented. This algorithm is discussed in more detail below. For the moment it's assumed that a method is in place to copy, delete, and/or rename files throughout the entire DMS.

In our preferred embodiment Step 22432 is usually done with a file copy for a Put operation, but the system does support an environment where files can be sent, or otherwise transmitted. To accommodate this, a special option called "ViaCopy" exists on the main promotion menu. It defaults to an "on" position, but can be turned off to signify that the data files are being sent to the ALM's reader along with the control information. If the files are sent, they must be "read" into a temporary holding area while the promotion process takes place. Otherwise, they can reside in the user's private space until the actual transfer takes place. At this point the file is physically copied from the source location to the Entry Level. In addition, any subordinate files that belong to this file's Automatic File Group are also transferred to their destination.

For a promote, the preferred method is to move the file(s) from the source location to the destination location. This may result in a simple rename of the file(s) if both physical

locations are identical, or it could result in the file(s) being physically moved from one server to another. The exception to this is if the Retain Source option is specified on the promotion screen in Step 22101. This would result in copying the file(s) from the source to the destination.

For a BOM Promote, a list of all BOM members is returned by the Control Repository in Step 22425. All members on this list are moved in the same manner as the Anchor file indicated in the main File List.

Similarly, if the current file is the Master of a File Group, a list of all subordinates are returned from the Control Repository during Step 22425. This list is used to move all the subordinates to their target destination.

For environments incorporating Automated Library Machines, the ALM must have a means to access and update any data within its own library. Whenever possible, all ALMs in a given library are provided with read and write authority to all physical data repositories. For example, in a simple DMS all data within a library would reside in a single directory, and the ALM would have read and write authority to that directory. However, since our embodiment permits different PFVLs to reside in separate repositories, this can't always be achieved. For instance, one PFVL may reside in a Unix or AIX directory, while another PFVL resides on a VM Minidisk. An ALM running in the Unix/AIX environment may not have direct write access to the VM Minidisk. The following algorithm is used to handle all types of file movements in the DMS, depending on the ALM configuration and platform environment.

In order to maximize efficiency, the ALM will always try to rename or move a file if the environment supports such an operation and the source and target PFVL reside in an amenable environment such as two Unix/AIX directories (same or different) or the same VM Minidisk. In this case, the ALM attaches to the target repository in a writable manner, and performs the move operation. Otherwise, the operation is performed by a combination of file copy followed by file deletion. In this case, the algorithm first determines if both the source and target repositories are writable by the ALM. Such is the case for a Conventional ALM system or an Actor/Object system running in a Unix/AIX environment. Here, the ALM directly performs the copy from the source to the target PFVL. Once the file is safely copied, the ALM deletes the source file. In an Actor/Object configuration in a VM environment, the Actor must send a message to the Object and pass a list of files to be copied and deleted. The Object runs a continuous message handler which allows multiple Actors to interrupt and initiate the copy and deletes. For situations such as a Conventional ALM arrangement running on a VM system, where the source and target PFVLs are on different account Minidisks, the ALM handling the promotion always has write access to the target PFVL. Therefore, it can directly perform the copy operation. The delete is handled by generating a file deletion request similar to that generated by the File Deletion routine. This job request is transmitted to the ALM with write authority for the source repository.

For promotions involving the movement of data across different platforms, our embodiment incorporates the concept of a Cross-Platform Transfer. The method differs depending on the source and target platforms. Our embodiment will always try to use the ALM currently executing the Promotion algorithm to link to the target platform directly. In these cases, the underlying Actor/Object code sets up the appropriate file transfer protocol and copies the file from the source to the destination. It then deletes the file from the source repository. The most complex case is when the target

environment can't be linked directly by the ALM. Such is the case when the ALM is running in a Unix/AIX environment, but the target PFVL is on a VM Minidisk. Achieving this file transfer involves using dedicated ALMs running on the source and target platform, which serve as agents to forward the work requests. The following steps are performed:

1. The ALM prepares a special Cross-Platform Transfer request file which contains the source and target level, the userid of the requester, the type of promote, and a list of all files that need to move. This request file and the original promote request are copied into a special punch directory whose name is also contained in the request file.
2. The ALM currently processing the promote, establishes a socket connection with the VM agent to request a file transfer to VM.
3. The VM agent copies all the data from the punch directory, and the Cross-Platform Transfer request file to its working space.
4. The request is sent to the appropriate VM Actor, where it's handled by Steps 29162 and 29163 of the ALM algorithm. This algorithm is described in detail in FIG. 55c. Step 29163 takes care of transferring all the data as well as deleting the copies from the source repository.
5. At this point, the code exits the Promotion algorithm, and the remaining steps in FIG. 15d and 15e are performed by the code executed in Step 29163 of the ALM algorithm.

Note: Our embodiment only supports Cross Platform promotions using Actor/Object arrangements.

An alternate embodiment permits all the files in the DMS to reside in the same physical location. Symbolic links would serve as place holders and reside in the locations defined by the Data Manager for each PFVL. During a Put, the link would be created while the file is being copied to the master location. During a Promote, the file would be renamed and the corresponding link would be updated. Depending on the environment, this may provide performance or data maintenance advantages over the preferred embodiment.

The last step within the File Loop pertains to the Update and Overlay Locks established in Steps 22414 and 22417. In Step 22433, Rst Lock, the Reset Lock Option is examined. If it's on, this signifies the user's desire to leave the file in an "unowned" state at the completion of the Put or Promote, and the program resets any Update Lock that exists for this file at the new level. For promotes, an additional step is taken to reset the temporary Overlay Lock set in Step 22414 regardless of the Reset Lock Option.

Control is returned to Step 22412 in FIG. 15c until all files in the request are exhausted. Upon exit from the loop, control is passed to FIG. 15e. Here Step 22427 is again invoked to send a message to the user indicating a successful Put or Promote. The message includes the names of all files processed including Automated File Groups and members of a BOM.

Next, the DMS again invokes Step 22421, Process Manager to run any Post-Processes that exist for any of the files. Just like with Pre-Processes, the Process Manager handles the entire list of files with a single invocation.

That last steps in the operation attempt to determine whether additional promote requests are necessary. In Step 22434, Prom Req'd, the Destination Level is compared against the current level for each file in the list. For Puts, the

current level is the Entry Level. For promotes, it's the Source Level. If any files are found with a Destination Level higher than the current level, this indicates that further promotion is necessary, and Step 22435, Promote, is invoked to handle this. In this step, a promote request is actually created listing all files which require further movement. The control file also lists their newly acquired level as their source level and lists their Destination Level as the target level. The entire background algorithm is repeated with Step 22411, but using the newly created promotion request as the main input control file. Once the current level and Destination Level are the same, the promotion is complete.

Methods of Storing, Retrieving, and Managing Data in a Shared Library System

The present embodiment incorporates various algorithms and processes to permit data to be stored into, deleted, and retrieved from the Data Management System while maintaining absolute data integrity. The Data Management System (DMS) is comprised of one or more shared public libraries servicing one or more users in a client server environment. The users may also have private libraries where work is performed until such time that it is ready to be deposited into a public library for shared access. Our embodiment has the capability to provide continuous access of the shared libraries to large numbers of users.

These algorithms and processes are especially suited to interacting with the File Promotion Process and Automated Library Machines (described in the other sections of the Preferred Embodiment), although they will work without Automated Library Machines and with other File Promotion Algorithms.

In order to safely deposit data into the Data Management System, our preferred embodiment uses an Installation Algorithm described herein. The algorithm is depicted in a library environment using Automated Library Machines configured in any arrangement, although one skilled in the art would appreciate that the algorithm can function in the absence of ALMs simply by executing in the client's foreground environment.

The most frequent initiator of install requests is the Process Manager. The Process Manager executes Automated Library Processes which create output data that must be deposited into the DMS. Our embodiment maintains close ties between the output data and the source data used to create it. If this output data can't be successfully installed into the DMS, the source data may be prevented from moving through the DMS or undergoing further processing until the problem is corrected. Because of the multitude of different types of Automated Library Processing, our preferred embodiment contemplates several types of install requests combined with a plurality of user options.

The following types of installs are supported:

Regular Install Deposits the output into the DMS under full data control. The output file may or may not be assigned a File Reference number and completely tracked by the Control Repository. Once installed, the file can be processed like any other file in the DMS. The install can be immediate, whereby the install is initiated and further processing suspends until it completes, or the install can be delayed which means further processing may continue while the Library Manager processes the install request.

High Performance (HP) Install Similar to a Regular Install, except it deals with groups of component files. This occurs during an Aggregate Install which always begins with an Anchor file undergoing a Regular Install, followed by all the related components under-

going a High Performance Install. The information necessary to install the group of files is conveyed through a special file transmitted with the Install Request.

Create DILP Install Used only on the output file of a Create DILP. This install is identical to a Regular Install with some additional steps that assign the process result to the output file once it's safely in the DMS. The necessary information is passed to the Install algorithm through a special file transmitted with the Install Request.

Regular Store Deposits the output into the Data Repository, but the file is not tracked by the Control Repository. These files exist for some other reason than data management, and don't require any DMS operations (promote, library processing, problem tracking, etc.). This type of disposition can be immediate or delayed.

The following options are supported for the above types of installs:

Result: Allows Pseudo Process Results to be set against files being installed.

Note: All information is exchanged via a Results File

Part Number: Allows Part Numbers to be assigned to files being installed.

Note: All information is exchanged via a Part Number File

Resolve: Allows Library Process Results to be primed for files being installed as long as the process structure for the installed files is identical to that of the source files. For example, if a file is copied from a source PFVL to a target PFVL, since the installed file is identical to the source file, any Library Process results for the source file can be automatically propagated to the target file.

Note: All information is exchanged via a Resolve File

Now, the program begins by receiving an Install Request. The Install Request contains the target Version and Level, the e-mail address of the user initiating the request, the File Name, Type and Package of the subject of the install, the Level Reference Number, one or more File Reference Numbers representing the data which was used to create the subject data, a multiplicity of user options, and a flag indicating whether a new File Reference number should be acquired for the subject data or whether an existing reference number should be retained. This information is generated by the algorithm responsible for creating new data which needs to be installed in the DMS. A typical example would be our Background Automated Library Processing.

In a Conventional System, this request may be transmitted from a Remote Execution Machine to a Primary ALM whose responsible for installing the data. In this case, the Install Request contains additional information regarding the Process Manager's Process Queue. This includes the Library Processing Phase (Pre, Post or DILP), the File Reference of the file used to set the Processing Lock and the Process Queue Reference Number. A Checksum or Cyclic Redundancy Code is also included to help detect transmission errors in the data. In an Actor/Object arrangement, the Actor can execute this algorithm directly, so there's no need to transmit the request. In this situation, no Process Queue entry is required and no CRC code is needed. However, the Install Request file is still created.

The algorithm begins with Step 24910, which Reads the Request File into a data structure. In addition any user options are examined and for those requiring separate support files, the file names are assembled. The program also

determines whether the ALM currently executing the algorithm is an Actor, and if it's not an Actor it checks to see if the request was sent from a different ALM. If so, the information in the Install Request is written into an Install Recovery file which is used by the ALM Algorithm to recover the install in the event of a system crash. Also, in the case where the Install Request is transmitted, the subject data is also transmitted, so it is transferred into the ALM's working space.

Next, Step 24911 is invoked to perform a Data Access of the physical repository where the subject data will be deposited. This may be as simple as ensuring the ALM has the proper write authority (such as a Unix/AIX environment) or it may require linking media such as DASD in a VM environment.

Step 24912 tests for Return Code=0 from the previous step. A non-zero return code indicates that one or more physical repositories could not be located or accessed in the proper manner. This results in Step 24913 being called to possibly clear the Process Queue. Step 24913 tests to see if the Install Request was transmitted by a Remote Execution Machine. If so, the Process Queue Reference Number is used to delete that entry from the queue. Furthermore, if an entry exists and the Process Phase is a Pre-Process, then a special entry is added to the Process Queue in order to prevent any subsequent Library Processes or movements occurring against the source file used to generate the subject file being installed. This maintains data integrity in the DMS by ensuring that output data always matches input data, and input data may not be elevated to the next level unless all required Library Processes run successfully and the output is successfully deposited into the repository. Upon completion of Step 24913, the requestor is notified of the failed installation attempt and the program exits.

Returning to Step 24912, if the Return Code equals zero, then the physical repositories are ready to accept the data and control proceeds to Step 24914. In Step 24914, the Actor flag, set in Step 24910 is examined. If the flag indicates that the current ALM is not an Actor, then the Install Request was transmitted with a Cyclic Redundancy code which pertains to the data being installed. The program generates a new CRC Code using the subject data in the working area and compares this with the code embedded in the request file. If they don't match, a transmission error occurred which is a data integrity violation. This results in a message being returned to the requestor, and the program aborting.

Assuming the transmission occurred successfully in a Conventional System, or if the current ALM is an Actor, then control continues with Step 24916 which sets the Fix Management Flag. This flag is passed to the underlying QRSUPGEN routine to tell it whether to propagate the Problem Fix Management data from the source files to the subject file. This flag is set to "true" if the Package is in Single Fix Tracking or Engineering Change Mode and the Library File Type has its Fix Management flag active.

At this point, control is passed to Step 24918. In the preferred embodiment, Step 24918 is exercised before Step 24922, but these are independent tests which can be performed in either order. In Step 24918, the Create DILP option is examined. If the current request is for output generated by a Create DILP, then Step 24919 is executed to Read the Create DILP Info. This information is stored in a separate file which accompanies the Install Request and the subject data. The file contains the true name of the output file, which may differ from the name contained in the Install Request. Create DILPs are part of our library process. This file also contains the Log and Process Reference Numbers of

the Library Process which created the file, and the Library Process Return Code. Step 24920 is then invoked to Create Recovery Information. This consists of generating a Create DILP Recovery File which includes all the information in the Create DILP file along with the corresponding Control Repository command required to store the information in the DMS. In the event of a system crash, the ALM Algorithm will use this file to exercise all the steps necessary to recover and complete the installation.

Finally, Step 24921 performs a File Copy into the repository. The file is copied here temporarily so it's sure to exist in the event of a crash. However, if the subject file is overlaying an existing file, the existing file is backed up first. This enables the ALM Algorithm to completely and accurately reproduce the state of the DMS if the subject file is unable to be properly installed. A flag is written into the recovery file indicating whether an overlaid file has been backed up.

If the current request is not for a Create DILP, then Step 24922 is employed to test for a High Performance install. This option can be requested to install a group of files. Our embodiment requires that at least one member of the group to act as an Anchor file. This means it must endure all the normal checking of a regular install. The remaining files benefit from the High Performance install which eliminates certain checks that are redundant to those done on the Anchor file. The list of files to be installed is maintained in a separate file transmitted with the Install Request. Step 24923 Reads the File List into a data structure that will eventually be passed to the Control Repository.

Step 24924 is then invoked to Access All the Repositories necessary to accommodate the entire list of files. Since our embodiment permits the list to include files of differing PFVLs, and files may be physically segregated down to the PFVL granularity, a plurality of repositories may have to be acquired. The method of access is identical to that performed for the subject file in Step 24911.

Whether the tests performed in Steps 24918 and 24922 are true or not, control eventually reaches Step 24928 which Updates the Control Repository. This step consists of updating all the necessary tables with the information required to track the subject file(s). In the case of a Create DILP or regular install, the QRSUPGEN routine, is called. In addition to the File Name and PFVL information, the algorithm also passes the New File Reference flag, the Problem Fix Management flag, and any of the Source File Reference numbers pertaining to the source data used to create this data. In the case of a High Performance install, the same information is passed with the exception of the Problem Fix Management flag. Instead a list of the entire file group is passed. If an error occurs, such as the existence of a lock which prevents the install, and this is a Create DILP which necessitated the back up of an old subject file in Step 24921, then the backed up file is restored. All severe errors during this step result in a message being sent to the user, followed by the immediate termination of the installation.

If no severe errors occurred during Step 24928, then control proceeds to Step 24930 in FIG. 33c. Here the return code of QRSUPGEN is examined for a Bill of Materials (BOM) Invalidation Message. Two types of warnings exist. One case is when the subject file overlays an existing file which happens to be the Anchor of a BOM. In this case, the BOM is deleted and the owner of the BOM is notified. The other case is when the subject file overlays a file which is the member of a BOM. In this case, the BOM becomes invalid and the owner of the BOM is again notified.

Next the program again invokes Step 24918 to test the Create DILP option. If this is a Create DILP install, then Step

24932 is employed to store the Process Result into the Control Repository. This result, along with the Process and Log Reference Numbers are contained in the Create DILP file. Since a Create DILP requires the Library Process result to be recorded against a file that doesn't exist when the process completes, the Process Manager must forfeit the setting of the LP result to the Install algorithm. Once the result is set in the Control Repository, Step 24933 checks to see if an overlaid backup file exists from Step 24921, and if so, it Erases the Backup file. It also erases the Create DILP recovery file, since all the steps that can be recovered have succeeded.

Control eventually reaches Step 24934 which Deposits the File into the data repository. The program checks to see whether the current ALM is an Actor. If so, and the environment allows the Actor to have direct write authority to the repository (such as Unix/AIX), the ALM copies the subject file from the Actor's working space to the target repository. If it's an environment such as VM, then the Actor establishes a communication link with the corresponding Object ALM, and requests the file copy. If the current ALM is not an Actor, then it must have write authority to the repository, so it simply copies the file from the working space to the physical location. If the current install request is for a High Performance install, the same steps are run against each file in the file list except for the Anchor. The Anchor is bypassed since the definition of a High Performance install requires the Anchor to be installed separately using a regular install request.

Next, Step 24935 is performed to test for the Result Option. If this option is passed, then there are Pseudo Process results which need to be recorded against the subject file. Step 24936 is employed to Read the Results File which contains the Pseudo-Process Name, the result, an optional message, and the name and PFVL information of the file for which the result should be recorded against. The code looks through the file for a matching file name and PFVL, and if one is found, it calls upon Step 24937 to Set the Pseudo. This is done via a function in the Process Manager to set the Pseudo-Process result into the Control Repository. Disclosure # PO996-0009 contains more information on Pseudo-Processes and the Application Program Interface that allows other routines to set them.

Eventually control passes to Step 24938 which tests for the Resolve Option. This option allows Library Process results from a different PFVL to be recorded against identical processes for the subject PFVL. If this option is passed, Step 24939 Reads the Resolve File which contains the Process Name, Process Reference Number, PFVL information where the source process resides, the Process Result to be recorded, the File Name and PFVL of the subject file and an optional Process Message.

Step 24940 is then employed to Find the Matching Process. This is done by querying the Control Repository for all processes defined for the source PFVL. The code loops through the process definitions until it finds one whose Process Name and Reference Number match the information in the Resolve File. If no match is found, the information in the Resolve File is inaccurate, and the program terminates with a message sent to the user. If a matching process is found, the exact location in the process structure is saved. Next, the control repository is queried for the process structure defined for the subject PFVL. The same location in the process structure is examined to ensure the Process Name is identical to that listed in the Resolve File. If not, then the structures are different, and the rule regarding use of this option is violated, thus causing the process to abort.

If the Process Names match, the Process Reference of the subject PFVL is used to query the Control Repository and store the Process Result and Message against the file being installed. Step 24941 interacts with the Process Manager to Store the Result into the Control Repository.

Eventually, Step 24942 is invoked to test for the Part Number Option. This option is used to store PIN information against the subject file. If this option is passed, Step 24943 is called to Read the Part Number File which contains the Part Number, the File Name and PFVL for whom the Part Number belongs, and the owner of the P/N. The program loops through the Part Number File until it finds an entry whose File Name and PFVL match the subject file. Step 24944 is then employed to interact with our Release Control Manager, to Assign the Part Number information into the Control Repository.

Note: In the case of a High Performance install, the subject file referred to in Steps 24935 through 24944 would actually be all files in the High Performance File List whose names and PFVLs match those listed in the Results, Resolve and Part Number files.

Finally, Step 24913 is executed in the event that a Process Queue entry requires clean-up. If the install request was received from an ALM other than the current ALM, then the sending ALM set an entry in the Process Queue. That Process Queue Reference number is included in the install request, and is used to remove the entry from the queue. In addition, if the Process Phase is a Pre-Process, and any part of the algorithm failed to complete successfully, then a special entry is added to the Process Queue to prevent any subsequent Library Processes from executing against the source files. In addition, the source files are prohibited from moving through the DMS until the install can be successfully retried.

File Check Out Utility

Our embodiment permits the user to request ownership of any file in the DMS and associate that ownership with an entry point into the library. This concept of ownership by entry point allows a sophisticated environment to exist where one owner can modify data at one level, promote the data to a higher level, and a second owner can make further modifications. The DMS Architecture also permits a user to set Update Locks at non-entry Levels, or set locks on ALL Levels which prevents another user from modifying or moving the file. However, these actions are not part of the File Check Out Utility and must be performed through a lock setting utility in the Lock Manager.

The user is also given the option to physically copy the file from a public library into their private library. The entire operation runs in the user's environment without the aid of an Automated Library Machine. The request will be honored or rejected depending on the current state of ownership and the relationship of the user to the current owner. The user initiates the operation with the File Check Out menu.

The preferred embodiment presents the user screen in a graphical environment where the user engages pull down menus, pop-up menus, drop-down lists, radio buttons, push buttons, fill-in fields, and mouse interaction. It should be noted, however, that all functions discussed further in the preferred embodiment can be implemented using simple text screens, or more advanced data entry systems such as touch screens, voice commands or 3-D graphics. The preferred embodiment depicts the method most conducive to the Motif™, Windows™, and OS/2™ application environments.

The screen contains six data entry fields, labeled 25901 thru 25906. Field 25901 is where the user types in the name

of the file. Fields 25902 thru 25904 are used to denote the Library Name, Library File Type and Version in which the data permanently resides. If the file is currently in a private library, the user enters the name of the public library to which the file will be promoted to in the future. Drop down menu buttons 25907 can be used to display a list of all the known public libraries in the DMS. Button 25908 will display a list of the valid Library File Types used in the library. Likewise, button 25909 will show all valid Versions for the given library. If no library name is entered, then clicking on either button 25908 or 25909 will produce an empty selection list.

Field 25905 represents the Starting Level for the library search engine to conduct the search. This doesn't mean the file must exist at this Level, it's simply where the user desires the search to begin. This can be any valid Level associated with the Library entered in field 25902, or it can be the keyword user. This keyword instructs the search engine to first inspect the user's private library for the file. If it's not found there, then the search engine should traverse the library structure beginning with the Entry Level denoted by field 25906. Drop down menu button 25910 can be used to acquire a list of all available Levels for the Library, Version, and Type entered in fields 25902 thru 25904. One of the choices is always the keyword user.

The Entry Level field, 25906, serves two purposes. The first is to provide direction for the library search engine in the event that field 25905 indicates user, but the file is not found in the user's private library. In this case, the search engine will begin at the level entered in field 25906 and traverse through the library structure until the file is located. The second purpose is to determine which level the Update Lock is to be associated with. Our embodiment permits multiple users to hold Update locks on the same file, but at different entry points into the DMS. Button 25911 displays a list of all valid Levels for the given Library, but unlike field 25905, the keyword user is not permitted.

The algorithm for checking data out of the library begins with Step 25951, SLL=User. Here the Starting Library Level entered in field 25905 is checked to see if it's the User Level of a private library. If so, the user's private library is examined to see if the User File Exists in Step 25952. If so, the Lock Check subroutine illustrated in is invoked and the program completes. The Lock Check routine is discussed in greater detail later.

If the file doesn't already exist in the User's private library, then the SLL Library Search is employed. The standard library search engine is used to seek out the most recent copy of the file starting at the User Level. The search engine also uses the Entry Library Level entered in field 25906 of to direct the search through the proper entry point. If no file is found, the user is asking for an Update lock on a non-existent file which is an error condition that terminates the program. At the conclusion of the search the user is shown the solution of the search. The Lock Check subroutine is again used to establish an Update lock. Upon return from the Lock Check routine, Step 25954, File Copy, is invoked. The program asks the user for permission to copy the file from the Library Level into the User's private library. The file is renamed to the SLL, which is user in this case, and the program completes.

Returning to Step 25951, if the SLL is not the User level, it's assumed to be a valid Level for the Library, Library File Type and Version entered on the menu. This Starting Library Level is used to initiate the library search engine. Upon completion of the search, the solution to the search is shown to the user. The Lock Check subroutine is invoked, and upon

return, Step 25955 is employed to see if the Starting Library Level File Exists. If not, Step 25954 is again invoked to copy the file to the user's private Library and rename it to the Starting Library Level. The user is given the opportunity to confirm this operation.

If a file already exists with the same Starting Library Level, the program indicates this to the user, in Step 25956, by showing the solution of the search along with the file in the user's library. The user is given the opportunity to replace the private copy of the file with the library copy. If the user accepts it, the file is copied and renamed using the Starting Library Level. If the user rejects it, the program terminates with the end result being an Update Lock set on the existing file in the private library.

Turning to the Lock Check Subroutine, the algorithm begins with Step 25960 which calls upon the QRSUPGET to return all the lock and authority information about the file. Next, Step 25961, EEL-LOCK-examines these locks to see if any are owned by someone other than the user. If so, these other locks are displayed so the user can see who else claims ownership of the file. If another user has any ownership locks (at the same or different level), the user is given the opportunity to abort the check out. In addition, the user is also notified whether they have the proper authority to promote this file into the desired Entry Library Level.

If a lock exists for the Entry Library Level, it's checked in Step 25962 to see if the User Owns It. If so, the user officially owns the "key" to this "entry door" and the routine passes control back to the main algorithm User Surgte, is invoked. Here, the database is queried to see if the user is a valid surrogate for the current owner of the lock. If not, then the user is told why the lock can't be set in their favor and the program terminates. If the user is a valid surrogate, then Step 25964, Reset/Notify is employed. In this step, the user is told who currently owns the lock and is given the opportunity to take ownership. If the user accepts it, the DMS sends a notification to the previous owner indicating that the user has now taken ownership of this entry point. The routine returns control to the check out algorithm.

Returning to Step 25961, if no Update lock corresponding to the Entry Library Level exists, then an Entry Library Level Lock is Set in Step 25965 for the user. This is done via interaction with our various Loch. Manager functions.

At this point the program returns control to the main algorithm.

File Deletion Utility

Since data integrity can be easily compromised by uncontrolled file deletion, our embodiment provides a robust utility for deleting data in a safe and orderly manner. It also ensures that the control information such as Problem Fix Numbers and Part Numbers are correctly handled. The preferred embodiment depicts the overall flow of the delete or data removal.

The flow begins with Step 28101, Entry Screen, where the user is presented with the File Deletion Screen. This screen, permits the user to enter information about the file(s) they wish to process. In Step 28102, Foreground, the information gathered in Step 28101 is processed and additional information may be requested. Some basic checks are performed before passing control to Step 28103. In Step 28103, Background, our preferred embodiment processes the request on an Automated Library Machine since these are the only users with the proper permission to edit or delete data within the DMS.

The preferred embodiment presents the user screen in a graphical environment where the user engages pull down menus, pop-up menus, drop-down lists, radio buttons, push

buttons, fill-in fields, and mouse interaction. It should be noted, however, that all functions discussed further in the preferred embodiment can be implemented using simple text screens, or more advanced data entry systems such as touch screens, voice commands or 3-D graphics. The preferred embodiment depicts the method most conducive to the Motif™, Windows™, and OS/2™ application environments.

The screen contains five data entry fields, which could be labeled 28211 thru 28215. Field 28211 is where the user types in the Name of the file. The user may type the name in directly or leave it blank to generate a selection list which allows the user to choose multiple files to delete. Field 28212 denotes the Library where the file(s) reside. This function is only intended for data tracked in a public library, therefore this field must contain the name of a valid public library. Drop down menu button 28216 can be used to obtain a list of all the public libraries in the DMS.

Fields 28213 thru 28215 are used to enter the Library File Type, Version and Level where the file(s) reside. Button 28217 will display a list of the Library File Types used in the Library, button 28218 will show all valid Versions and button 28219 will display all valid Levels. This information is used to initiate a library search for the file specified in field 28211. In the event the file doesn't exist at the specified Level and Version, a dialog box will display the closest file found in the library search. The user is given the opportunity to accept or reject the result of the search. If field 28211 is left blank, a selection list resulting from the library search will be displayed and the user may select as many files as desired.

In the cases where the file being processed is under Part Number or Problem Fix Control, the user can explicitly specify the Level and/or Version of the previous file which should be used to reassociate the Part Number and/or reactivate the Fix Management data. The Level and version are entered in fields 28220 and 28221 respectively. These fields are optional, and if left blank will cause the Foreground process to interact with the user to obtain the information. Drop down menu buttons 28222 and 28223 can be used to show a list of valid Levels and Versions for the corresponding Library.

The only option for this operation is the Model option which the user can specify via push button 28224. This is the means by which the user acknowledges that deletion of the file will cause either Bill of Materials deletion or invalidation.

Returning to the overall flowchart, information entered in Step 28101 is now passed to Step 28102, Foreground Processing. The detailed algorithm for generating file deletion requests begins with Step 28311, Parse Opts. Here all the options are examined to ensure they are recognized and the values are acceptable. If Previous File Info is passed as an option, the values are checked to ensure they exist for the given Library.

In Step 28312, a File Loop is initiated in the event the user selected multiple files from the user screen in Step 28101. In this case each file must be subjected to the same checks and tests since each file possesses individual characteristics.

The next series of steps pertain to handling Bill of Materials (BOMs), if they exist. Beginning with Step 28313, Model Opt, the program tests the Model Option flag. If this flag is true, it indicates the user accepts the possibility of BOM Deletion or Invalidation and does not wish to be warned in advance of the consequences. One example is the act of deleting a file which is an anchor to a BOM. If the user knows this in advance, they can pass this option to avoid unnecessary checks. In this case control proceeds to Step 28317.

However, if this option is absent, Step 28314 is invoked to check for BOM Deletion. Here, the Control Repository is queried to see if the current file is a BOM. If so, Step 28315, Warn User is employed to notify the user of the impending BOM deletion. The user is given an opportunity to abort the process. The next step, 28316, queries the repository for any BOM Invalidation. This tests for the situation where the current file belongs to some other BOM, so its removal will result in a BOM becoming invalid. Our Aggregate Manager is used to quickly locate any BOMs in the DMS which contain this file.

Once again, if an invalidation will occur, Step 28315 is employed to notify the user and give them the chance to abort the deletion.

Regardless of the setting of the Model Option flag, control eventually proceeds to Step 28317, File Check. In this step, the algorithm queries the database to ensure the file exists in the Control Repository and the repository agrees on the Level and Version. If the Level and Version returned by the Control Repository are not identical to that indicated by the user in Step 28101, an error occurs which notifies the user and aborts, the program.

The algorithm proceeds to Step 28318 where the Part Number and Fix Management Flags are obtained from the Control Repository and examined for the Library File Type being processed. If the LFT is under Part Number Control, Single Fix Tracking or Engineering Change Mode, control proceeds to Step 28319. In Step 28319, PN/FM Info, all Part Number and Fix Management information is obtained for the file being processed. In addition, all obsolete files, of the same name, and at higher levels, which are attached to the same part number and/or attached to the same EC Number are also returned.

At this point the algorithm determines which dormant file, if any, will be used to reassociate the Part Number and/or reactivate the Fix Management information. First, the Previous File Info fields 28220 and 28221 are examined in Step 28320. If the user specifies a particular level or version, they are expecting the corresponding file to be used to reassociate the PN and/or revalidate the problem fix numbers. In this case, the program employs Step 28321, Locate Previous File, to sift through the data returned in Step 28319 in search of the expected file. If the file is not in the list, it's an error condition and the program terminates. Assuming the file exists, the program proceeds to Step 28322, Trap PN/FM Info. In this step, the algorithm sets a flag and remembers all the information necessary to disassociate the old Part Number and Fix Management data from the file about to be deleted. It also captures the information about the file selected for re-association. Although the information is captured, the actual updating of the Control Repository is done in a later step.

If the Previous File Info fields are empty, the programs checks to see if the list returned in Step 28319 only contains a single entry. If so, Step 28315 is invoked to warn the user that the file in the list will be the one used for PN/FM reassociation. It also provides an opportunity to abort the process. If the user accepts this, Step 28322 is employed to capture the information.

The last possible case for PN/FM re-association, Step 28324, involves a list with >1 File being returned in Step 28319. If this is the situation and no Previous File Information is provided, the program uses Step 28325 to present the user with a Selection List. The user may select only one entry or abort the operation. Assuming one is selected, Step 28322 is invoked to capture the information.

If the Part Number and Fix Management Flags in Step 28318 are off, or no files were returned in Step 28319, or any

PN/FM information was trapped in Step 28322, control proceeds to Step 28326. In this step, the Level of the file being deleted is checked to see if it's a Release Level. This includes active or frozen Release or Sideways Levels. If the Level is any of the aforementioned types, Step 28315 is invoked to warn the user and provide an opportunity to abort.

Control eventually proceeds to Step 28328, PN/FM Re-assoc. In this step, the algorithm uses the information trapped in Step 28322 to interact with the Control Repository. It eliminates all Part Number information associated with the file about to be deleted, and reincarnates all Part Number information pertaining to the file found in the previous step. In addition, the superseded Problem Fix numbers attached to the file are converted to an active state. All appropriate Part Number and Fix Management tables within the repository are updated to reflect a state whereby the previous file assumes the role of the file being deleted.

At this point, control returns to the top of the File Loop in Step 28312. Once all files have been processed through Steps 28313 thru 28325, control proceeds to Step 28329, List Files. Here the Filenames, Library, Level, Version, Library File Type, and File Reference numbers are written into a Library Delete List which will be transmitted to the Automated Library Machine in the next step.

In step 28330, Xmit, the program gathers and transmits all of the necessary data to the Design Control System. In our preferred embodiment, the destination would be an Automated Library Machine which would "receive" the information from the user via an AutoReader. The following information need to be transmitted in the delete request:

The type of request: Delete

The list of files being promoted. The following information must exist for each file in the list:

- The Filename
- The Library File Type
- The Package
- The Version
- The Level

Any user selected options that pertain to the background operation.

The user's electronic id or e-mail address.

This file is transmitted to the ALM for use in the Background Processing step.

Returning to the overall process, the foreground information in Step 28102 is transmitted to an Automated Library Machine for background processing. The detailed algorithm for Step 28103, Background, begins when the algorithm enters a File Loop, in Step 28411, where the list of files transmitted from the Foreground are processed into a data structure. Each step in this algorithm must be performed against every file in the list.

Step 28413 obtains the Lock Information for the file from the Control Repository. This includes information about every possible lock the file possesses at any Level within this Version. In the subsequent steps, the list of locks are examined and different actions are taken depending on the types of locks in existence.

Step 28414 checks to see if any Processing Locks exist on the file at the specified Level. This would indicate a Library Process is currently dependant on the existence of the file, so Step 28415 is invoked to Recirculate the delete request. In our preferred embodiment this entails re-writing the delete request file with the names of all the unprocessed files, and sending it back to the main queue of the DMS. In a simple system without Automated Library Machines, the necessary action would be to introduce the request back into the DMS

queue or inform the user to try again at a later time. At this point the processing is complete.

If no processing locks exist, Step 28416 checks for Move or Overlay Locks at the Level where the file exists. In either type exists, Step 28420 Sends an Error Message to the user indicating that the file can't be erased. The program terminates after the notification is sent.

If no Move or Overlay Locks exist the program proceeds to where Step 28417 examines any Update Locks that exist. In this step all Update locks are examined regardless of the Level. In Step 28418 a determination is made as to whether the User Owns All the Update Locks. If this is true, then the user is the official owner of the file according to the rules of the DMS. In this case, control can proceed to Step 28421.

If there are some Update Locks which the user doesn't own, or no Update Locks exist at all, then the program checks to see if the user is the Package Manager or Alternate in Step 28419. As long as the user is the Data Manager or a valid alternate, the program is allowed to proceed to Step 28421. If the user is not a Data Manager, Step 28420 is invoked to send an Error Message indicating the situation, and the program terminates.

Assuming that the user meets one of the authority criteria, control proceeds to Step 28421 where the File is Checked to ensure the Control Repository agrees that it exists at the specified Level and Version, and ensure the file doesn't reside in a frozen Release Level.

Next, the algorithm checks the Fix Management Flag in Step 28422. This consists of querying the Control Repository to see if the FM flag exists for that Library File Type. If so, Step 28423 is invoked to Delete the Fix Management Information pertaining to the file. This is done via our Fix Management routines.

Step 28424 performs a similar function with the Part Number Flag. The repository is queried to see if the PN flag exists for the LFT being processed. If so, Step 28425 is implemented to Delete the Part Number Information pertaining to the file. This is done via our Part Number routines.

At this point control is passed to Step 28426 to Delete the Lock Information pertaining to the file. This is done via our Lock Management routines. If the user is a Data Manager, it is possible for the file to be in a completely unowned state. In this case, the DMS will not abort, but will continue with the next step.

In Step 28427, the QRFILDEL routine, described in FIG. 11, is employed to Delete the File from the Control Repository. This entails updating the necessary files tables to eradicate any associated entries.

Steps 28428 thru 28430 are designed to handle any Bill of Materials associated with the file. In Step 28428, the DMS checks to see if the file itself is the anchor file of a BOM. If so, BOM Deletion will occur for all BOMs associated with the file. Step 28430 is invoked to Notify the owners of all the BOMs about the elimination. BOM deletion is Performed via our Aggregation Management routines.

In Step 28429 the DMS checks to see if any BOMs are Invalidated by the removal of the file. If so, Step 28430 is again invoked to notify all owners of any affected BOMs. BOM invalidation is performed via our Aggregation Management routines.

The last step in the File Loop is Step 28431 which will Erase the File. This includes obtaining the physical location of the file from the Control Repository and performing the deletion. Depending on the environment, the Automated Library Machine may have the proper permission to perform a direct removal, or it may have to transmit a request to an agent which is capable of performing the removal. In our

preferred embodiment, the method of removal depends on the ALM configuration employed. In a Conventional System or any arrangement running on a Unix/AIX platform, the ALM can delete the file without assistance. However, in an Actor/Object configuration running on a system such as VM, or a complex system involving multiple computer platforms, the ALM may need to request the Object to perform the file removal.

Control returns to the top of the File Loop in Step 28411 until all files in the request are processed successfully. The operation then exits with a success message sent to the user.

Automated Library Machines (ALMs)

Our embodiment contemplates the use of Automated Library Machines (ALM) to process the work requests on behalf of the users. Although this embodiment is ideally suited for the various process and methods described in the other sections of the Preferred Embodiment, ALMs are not confined to running only those processes. ALMs may exist in Data Management Systems running processes and algorithms outside of those mentioned in this disclosure.

Our embodiment employs an Automated Library Machine (ALM) to service data management requests on behalf of the users. This enables the user to initiate a library job such as promotion request, Designer Initiated Library Process, or delete request without requiring significant client resources. The ALM provides continuous service by utilizing the concept of a reader to queue and prioritize users' requests. Performance of the Control Repository may also benefit since the most of the communication is with a relatively small number of ALMs compared to the larger number of individual users.

The preferred embodiment uses ALMs to execute all the Background algorithms included in the embodiment. One skilled in the art would appreciate that an alternate embodiment doesn't require ALMs by eliminating all the transmittal steps in the various Foreground algorithms, and simply running the Background algorithms on the client machines. As eluded to above, this may require substantial client resource and may compromise performance of the Control Repository.

For large Data Management Systems, our embodiment permits the creation of multiple ALMs to service a single library. This enables large user groups to redeem faster results through the use of parallel processing. The Data Manager has the option of arranging the pool of ALMs in one of three configurations depending on the expected type and volume of data management requests. The basic configuration is known as a Conventional System where a single ALM accepts all work requests and handles all services for the Library Manager, including any Automated Library Processing. The second configuration is Remote Execution Machines which is an extension of the Conventional System. Here, a single ALM receives all work requests from the user, and processes all promotion, installation, movement, and removal of data. However, additional ALMs may exist to perform Automated Library Processing. The ALMs interact with the Library Manager, Communication Manager and Promotion Algorithm to dispatch any desired library processing to a Remote Execution Machine, which executes the task and returns the results to the master ALM. The most powerful configuration is known Actor/Objects and this arrangement employs a pool of ALMs which serve as general purpose machines. They can perform any desired Library Management function, including Automated Library Processing. They can even interface with Remote Execution Machines to provide an environment with both general purpose machines and dedicated service machines. Each

Actor can be programmed by the Data Manager to define the type of work requests it can process. This arrangement even includes a special Dispatcher ALM whose sole purpose is to dispatch user work requests to the next available Actor machine.

The means by which data is physically moved between, added to, or deleted from, the repositories depends on the chosen configuration. In a Conventional System, the primary ALM is the only ALM with the proper authority to manage files on any repository within its own library. Remote Execution Machines may only receive work requests and return data and results to the Primary ALM. Our embodiment does not permit Conventional Systems to process file transfers across multiple platforms.

In this system, all actions are initiated by job requests. These may be include:

Class A: Requests to promote data from a user's private library into the public library or invoke Designer Initiated Library Processes.

Class B: Requests to promote data through the public library.

Class C: Requests for Automated Library Processing on a Remote Execution Machine or responses from Remote Execution Machines to indicate completed Library Processes.

Class D: Requests to install new data generated by Library Processes on Remote Execution machines, delete files from the DMS, or perform Data Management functions.

The classes represent priorities with Class D being the highest and Class A being the lowest. Every ALM in the library (Primary ALM and all Remote Execution Machines) runs the ALM algorithm as an AutoReader task. AutoReader automatically invokes the algorithm whenever a file is received in the reader.

In an Actor/Object system, either the Actors have the ability to directly manipulate files in the repositories (such as Unix/AIX), or they must rely on a dedicated ALM known as an Object to handle all file management tasks. The Object has the proper authority for all repositories in the library. The Actors run the ALM algorithm as an AutoReader task, just like the primary ALM in a conventional system. However, many of the Class C and D jobs related to file management are replaced by the Actor performing its own file manipulations (if the environment allows it), or by communication with the Object (such as a VM Actor/Object system).

In systems requiring an Object, communication between the Actor and Object is accomplished by an asynchronous messaging system where the Actor initiates a request to the Object and waits for a response message from the Object. The message consists of a command line which includes the:

Function to be Performed

Source File Name

Source File Type

Target File Name

Target File Type

Source Repository

Target Repository

The repository fields include enough information to physically locate the file regardless of the platform or environment. The Object, in turn, runs a continuous routine implemented as an AutoReader interrupt hook. Whenever it receives a message, the routine "wakes up" and checks to ensure the message is from a valid Actor, and contains one of the supported functions. It then executes the appropriate

function and sends a completion message to the Actor. If either the Actor or Object fails to transmit or respond to a message successfully, a mechanism will resend the message until the handshaking is complete.

Regardless of the environment, all Actor/Object systems support the following functions:

Rename Move or Rename the file from the source to the target location. This is only used on environments which support it such as Unix/AIX, or VM when the source and target are the same minidisk.

Copy Copy the file from the source to the target location. This is used to install an output file from an Actor into the repository, or as the first part of promotions involving a Cross-Platform file movement, Cross-Account or Cross-Minidisk file movement on VM.

Delete Delete the file from the source location. This is used for File Delete requests or as the second part of promotions involving a Cross-Platform file movement, Cross-Account or Cross-Minidisk file movement on VM.

Batch Used for multiple files which must be manipulated as part of the same task. A batch file is generated listing each file along with it's corresponding command line. The commands must be one of the three supported commands (Rename, Copy or Delete). The ALM loops through each line of the batch file and Processes each file successively.

Note: ALMs only deal with file manipulation requests. In the preferred embodiment, it's up to the DMS algorithm overseeing the file movement (such as the Promotion or File Installation algorithm) to determine which type of library arrangement exists and whether job requests should be created and transmitted to a Conventional ALM or Actor/Object commands should be generated and executed. Therefore, the underlying code for all these algorithms must query the DMS for the type of library arrangement. If it's Actor/Object, the code must also determine whether the environment utilizes an Actor/Object messaging scheme, or whether the Actor can execute the Rename, Copy and Delete functions directly.

Automated Library Machines are based on the concept of an Automated Reader where the Reader is a temporary storage area which accepts library requests. A Reader may simply be a directory where data is copied into, or it may be part of the environment such as the VM system. A simplistic implementation of an Automated Reader would incorporate a continuous loop with a timer to view the files in the Reader at specified intervals, and upon finding one, initiating the ALM algorithm. However, our preferred embodiment implements an Automated Reader by using an AutoReader service machine. This software machine is capable of performing many tasks outside the arena of Data Management, as well as providing the Automates Reader function.

Once a file is detected in the Reader, the ALM algorithm is invoked. It begins with Step 29120 where a registration check is performed. All ALMs must be registered with the Control Repository, and when registration is complete a flag is set. In Step 29120, Reg. Flag, this flag is tested to ensure it's set. If not, Step 29121 is invoked to Register the ALM. The ALM is first checked to ensure it's an authorized user of the Control Repository, and if so, it updates the repository with certain environmental information such as user id, system address, etc.

Upon completion of the registration, Step 29122 is executed to test for a Startup command. This is passed into the ALM algorithm as an AutoReader parameter whenever the machine is re-started. This could be the result of a system

crash or a manually initiated command. In the case of an ALM Startup, various tests are made to attempt to recover any interrupted or incomplete tasks. The first test, done in Step 29125, is for a Process Crash. This is done by looking for the existence of a Bucket in the ALM's work space. Our Process Manager writes a Bucket file each time it begins running an Automated Library Process. If the process completes normally, the Bucket is erased. The existence of a Bucket signifies a Mid-Process crash, which results in Step 29126 being executed to Send a Message to the user who requested the Library Process. This information is contained in the header of the Bucket file.

Control proceeds to Step 29128 to test for a Create DILP Recovery file. This file is created during the installation of the output of a special Automated Library Process known as a Create DILP. In the event of an ALM interruption, this file contains all the information necessary to retry the file installation. Next, Step 29129 checks for the File Existence of the Create DILP output. Assuming it's present in the ALM's work space, Step 29130 is invoked to Automatically Retry the installation of all Create DILP output. The installation entails calling the QRSUPGEN function to update the necessary files tables as well as calling QRRESADD to add the Library Process result.

The third test is for a regular Install Crash in Step 29132. This is accomplished by testing for the existence of an Install Recovery file. Like the Create DILP Recovery file, this file is written as part of the file installation algorithm to aid in automatic recovery. If it exists, the recovery action is determined by the existence of the Process Phase parameter in the Install Recovery File. Step 29134 tests for the Process Phase. If it is absent, the installation was not initiated by an Automated Library Process, therefore Step 29136 simply Recirculates the Install Request. If the phase does exist and the originator of the request is an ALM other than the current machine, then Step 29138 is executed to test if the Phase=Pre Process. If so, then Step 29140 will be invoked to call the QRPRQADD function to add a special entry to the Library Process Queue. This prohibits the file undergoing Pre-Processing from moving through the DMS, or executing any further Library Processes, until the installation can be performed successfully. Regardless of the phase, Step 29142, QRPRQDEL, will eventually be called to delete the entry from the Library Process Queue that was created by the Install Algorithm to prevent any file creation or movement while the installed file is in transit.

Once the appropriate recovery action is completed, or if none of the three types of recoverable scenarios are satisfied, control exits the algorithm and returns to the AutoReader machine.

Returning to Step 29122, if the current request is not a Startup then control proceeds to Step 29123 to Order the Reader. This step incorporates a combined algorithm to provide first-come-first-serve processing for non-Data Management requests, while ensuring Data Management jobs are handled in order of priority. First the file is examined to see if it possesses a higher priority than a library request. The type of request is also checked to see if it's a supported library request. All library requests contain a LIB keyword in the job type. If none of these conditions are satisfied, the program immediately returns control to the AutoReader algorithm to process the non-Data Management request. Otherwise, this is assumed to be a library work request. In order to maintain data integrity, all library requests are processed in order of highest to lowest priority. Step 29123 accomplishes this by sorting all reader files, which possess one of the four job classes, in descending priority order. If

multiple files contain the same priority, they are sorted by time from oldest to most recent. This yields the oldest, highest priority file. The program then determines if the type is a library request. If so, it will be processed, otherwise, the sorted list is searched until the oldest, highest priority library request is found. This ensures that the library requests are done in the proper order, but still permits non-library work to be intermixed.

Next, Step 29124 is invoked to Resolve the Sender of the library request. This entails reading the sender's id and electronic address from the header portion of the work request. At this point control proceeds to a Receive the File. Receiving the file refers to moving the file from the reader to the ALM's workspace so downstream programs can access it. These downstream programs may be the Promotion algorithm or an Automated Library Process, but our embodiment ensures all ALM's use identical work spaces, which are environment specific, so any downstream process can easily find the data. For example, in an aix/unix environment, a nomenclated subdirectory is used as the ALM's work space, whereas temporary DASD is used in a VM system.

Steps 29146 through 29163 are used to direct the library request to the proper algorithm. It can best be handled with a case or select statement, and the don't imply any order for these steps. Step 29146 tests for a Report Request. Our embodiment permits the user to send requests for various nightly reports. If Step 29146 tests positive, then Step 29147, Rpt is invoked to add the request to the nightly report queue file. At a pre-determined time, a service machine wakes up and processes all the requests in the queue.

Step 29148 tests for a Promote Request. These can be requests to transfer data into a public library from a private library, or move data through a public library. The data may be an individual file, a group of files, or an aggregate grouping. Regardless of the type of promote, control is passed to the Promotion algorithm in Step 29149. This algorithm is detailed in FIGS. 14 and 15.

Step 29150 tests for a Delete Request. These are requests to delete data from a shared library, and they can be initiated by the owner of the data or the Data Manager. Delete requests are handled by the Delete algorithm in Step 29151.

The case structure continues with Step 29152 which tests for an Install Request. This type of request originates from an ALM acting as a Remote Execution Machine in a Conventional Library System. Since the Remote Execution Machine can only execute Library Processes, but not manipulate data, it must send a request to the Primary ALM to store the data into the repository. In this case, control is passed to the Install algorithm in Step 29155.

Step 29154 tests for a Store Request. This is almost identical to the Install Request in Step 29152, except that the data is not tracked in the Control Repository. It's simply deposited into the data repository without any affiliation to the library structure. These requests originate under the same circumstances as Install Requests, but they are handled by the Store algorithm in Step 29155.

The Store Algorithm in Step 29155 simply consists of reading the file information out of the request file and determining exactly which repository the file should be stored on. This information is contained within the job request. Next, the code receives the file from the reader and copies it into the appropriate repository. Since the file is not tracked by the DMS, no queries are made to the Control Repository. Furthermore, the nomenclature on the file consists only of a Filename, Library File Type and Version. There is no Package, Level, or File Reference Number.

In Step 29156, the program checks for one of the many types of requests associated with Automated Library Processes. Due to the many different library arrangements supported by our embodiment, any given ALM may be playing the role of a Conventional ALM, Remote Execution ALM, or an Actor. This means that any ALM must be capable of receiving a job request to initiate an Automated Library Pre-Process, Post-Process, or a Designer Initiated Library Process (DILP). Additionally, it may receive responses from completed Pre, Post or DILPs. All requests related to Library Processing are handled by our Automated Library Processing algorithm in Step 29157.

Step 29158 is designed to handle requests which Create a Structure File. Our preferred embodiment uses Structure Files to supplement the Structure Tables in the Control Repository. These files contain a formatted list of all the Levels and Versions installed for this Package, their repositories, and the information linking the Level and Version tree. This permits many of the Data Management functions to reference this file instead of querying the repository, thereby increasing availability and possibly improving performance. In order to assure that these files are kept in sync with the Control Repository, any changes made by the Data Manager to the library structure result in a Create Structure File Request being sent to the library's main ALM. Upon receiving it, Step 29159 is invoked to Update the Structure File using the latest information in the DMS. This step extracts the structure information from the Control Repository and writes it into the Structure File with the proper format.

The case structure continues with Step 29160 which tests for an Authority Request. Data Managers may elect to use Authority profiles to generate a master list of authorized users for their Package. Every time this list is generated, it's sent to the master ALM for the Package, where upon receiving it, Step 29161 is invoked to Replace the Authorized Users List. This simply consists of copying the newly received file over the existing user list. Detailed information regarding Authority Profiles can be found in our Authority Manager.

Step 29162 tests for a Cross Platform data transfer such as a file being promoted from a Unix/AIX platform to a VM platform. Step 22432 of our Promotion algorithm, describes how files are moved from the source repository to the destination. In many cases the ALM running the algorithm has the proper access to perform the necessary file transfer functions without any assistance. However, cases such as this one, don't permit the proper access to the ALM on the source platform. Therefore, the source ALM suspends running the Promotion algorithm and uses a special ELM, running on the target platform, as a communication agent to forward a Cross-Platform job request to any ALM on the target platform capable of writing to the target repository. This ALM on the target platform receives the Cross-Platform Data Transfer job request which requires the special algorithm in Step 29163 to be invoked.

Step 29163 runs the Cross Platform Algorithm, which begins by reading the header line from the Cross-Platform job request file. Next a loop is established to process each file listed in the request. For each file the source repository is linked in a read mode, and the destination repository is linked in a writable mode. The appropriate file transfer protocol is established and the file is copied to the target repository. The copy of the file residing in the source repository is then deleted.

At this point the file movement is completed, so control returns to the top of the file loop until all files are moved to

their target locations. The code then executes the same steps in the Promotion Algorithm that would've taken place if the source ALM performed the file movement. These consist of Step 22433 in FIG. 15d and all the steps in FIG. 15e.

If the request keyword doesn't match any of the tests, then control is returned to the AutoReader and a message is sent to the sender indicating a nonsupported library request. It should also be noted that this structure easily permits additional types of library requests to be added. For example, other environments may require a type of library processing not discussed in the preferred embodiment. By simply assigning a keyword to that type of request, any algorithm or program can be exercised upon receipt of the work request.

On the other hand, if any of the supported algorithms are executed, they will eventually return control to Step 29170 to test for a Processing Lock. Some of the algorithms such as the Promotion and Library Processing algorithms may not be able to process the current request if it involves data currently locked in the DMS. In this case, the algorithms return a unique return code, which Step 29170 detects. If it tests positively, then Step 29171 is invoked to Recirculate the request. This entails placing the request back into the Reader with the same priority, but a current time stamp. If there are no other work requests in the Reader, then this request will continuously loop through the Reader until the Processing Lock is relieved and the appropriate algorithm can service the request.

In a DMS utilizing ALMs, all Foreground algorithms transmit job requests to the ALMs which execute the Background algorithms. In order to assist these Foreground algorithms in locating the proper ALM to send the request to, our embodiment provides the following means. The preferred embodiment contemplates the use of a Master Library Directory which retains information about every library in the DMS. The listing is sorted by Package ID, and each record indicates whether the libraries for that Package are running in Conventional mode or Actor/Object mode. Additionally, the record indicates the primary repository for that library. This repository holds any data with library-specific data such as Library Logs, AutoReader control files, Actor lists, etc. User data may or may not be located in this repository.

The Master Library Directory may be maintained within the Control Repository or as a separate flat file kept in a commonly accessible location. Since any authorized user of the DMS may invoke a Foreground algorithm, all users' client environments must have access to this information. Regardless of its location, the Foreground algorithms always follow this procedure for transmitting job requests. First, the Package is checked to see if it's running a Conventional or Actor/Object system. If it's Conventional, then the primary repository is the Primary ALM where all job requests should be transmitted. Using the four level Class system explained above, the Foreground algorithm directs the job request to the Primary ALM's reader queue. Eventually, the AutoReader accepts the job request and initiates an ALM to process it. If the Foreground algorithm detects an Actor/Object system, then it must locate the Actor List. This is a listing of all the Actors servicing a given library. For each Actor in the list, information exists denoting the type of work requests it's allowed to receive. The Actor List is established by the Data Manager using our Data Management Configuration Utilities. A utility exists which permits the Data Manager to easily define a multiplicity of Actors with their corresponding qualifications. This information is stored in the Control Repository and may be duplicated in a text file which is stored in the primary repository.

Once the information in the Actor List is acquired, the Foreground Algorithm looks for a special Actor called a Dispatcher. In systems without dispatchers, the algorithm simply scans the Actor List until it finds an Actor capable of accepting the type of work request being transmitted. If more than one work request is being generated, and more than one Actor is capable of accepting that type of work, the requests are distributed evenly between the Actors in a simple round robin scheme. However, our embodiment incorporates the use of a Dispatcher to maximize efficiency by receiving all work requests from all users into a single "Bank Teller" queue. The Dispatcher is a special purpose ALM whose sole job is to accept work requests from users client machines, and dole them out to the next available Actor capable of servicing that particular request. The Dispatcher ALM runs a special Dispatching algorithm which is described below. Once the work request reaches the Actor, it's handled in the same manner as a Conventional System, whereby the request is received and processed by the ALM algorithm.

Note: Work requests generated in the users' environment are never sent directly to Remote Execution Machines. The use of a Remote Execution Machine is specified for Automated Library Processes by the Data Manager. When a user initiates one of these special Library Processes, the work request is first analyzed by the Primary ALM, in a Conventional system, or an Actor ALM. The ALM algorithm decodes the work request and calls upon our Library Process Manager to direct it to the appropriate Remote Execution Machine.

Our embodiment further enhances libraries arranged in an Actor/Object configuration by contemplating the use of a Dispatcher ALM. The Dispatcher is a special purpose Actor which accepts job requests from the user and holds them until an Actor capable of processing that work request is available to service it. Throughput is enhanced by ensuring all Actors are kept busy whenever possible, and the workload is balanced across all of them. Configurations using a Dispatcher identify the userid in the Library's Actor List with a special entry denoting it as a Dispatcher. All foreground algorithms which generate work requests check for this entry upon detection of an Actor/Object configuration. If it exists, the job request is sent to the corresponding userid. Otherwise, the Actor List is examined for the first Actor in the list capable of servicing that request. The job is then dispatched directly to that Actor. If multiple jobs need to be dispatched, the jobs are distributed to all the Actors, capable of handling the task, in a round-robin fashion. This scenario may lead to an unbalanced workload among several Actors if the job requests have a large disparity in processing times. The Dispatcher is designed to eliminate this.

The preferred embodiment permits any ALM to act as a Dispatcher simply by configuring the Autoreader to run a special Dispatcher algorithm. It works on the premise that Autoreader permits three types of interrupts:

1. A new request arriving in the Reader causes a PreCheck interrupt.
2. A message or command may be used as an interrupt.
3. Autoreader's built in timer acts as an interrupt when it "wakes up".

The algorithm continuously monitors for any type of interrupts. Upon receiving one, it must check to see if it's one of the three aforementioned types. There are other types of interrupts, but they don't pertain to the Dispatcher function.

Our preferred Design Control Repository and System Methods (5.0)

Here we will discuss how our preferred Design Control Repository and System Methods can be used with various changes which need to be adopted for existing software in order to make use of that software in our new environment. In this area we will discuss the AFS environment. In order to be used commercially in the future, we believe that the current software, now insufficient for our complex needs, needs to be changed. Our PFVL structure and principles are adopted.

As one reads this, one will appreciate that we now have a data management system for file and database management which has a design control system suitable for use in connection with the design of integrated circuits and other elements of manufacture having many parts which need to be developed in a concurrent engineering environment with inputs provided by users and or systems which may be located anywhere in the world. Using our PFVL structure and process principles as the foundation for the architecture we provide a set of control information for coordinating movement of the design information through development and to release while providing dynamic tracking of the status of elements of the bills of materials in an integrated and coordinated activity control system utilizing a control repository which can be implemented in the form of a database (relational, object oriented, etc.) or using a flat file system. Once a model is created and/or identified by control information design libraries hold the actual pieces of the design under control of the system without limit to the number of libraries, and providing for tracking and hierarchical designs which are allowed to traverse through multiple libraries. Data Managers become part of the design team, and libraries are programmable to meet the needs of the design group they service. A control repository communicates with users of the design control system for fulfilling requests of a user and with data repositories of said data management control system through a plurality of managers. Each manager performs a unique function. Managers act as building blocks which can be combined in a plurality of manners to support an environment for suitable for multiple users of a user community.

As we review our concepts in greater detail, it will be seen that the present embodiment describes a Data Management System (DMS) which is composed of a suite of function managers and one or more projects (see FIG. 16—Items 10, 11, 14, 15 and 16). Each project is composed of a central Control Repository and one or more data repositories (see FIG. 16—Items 12 and 13) to store, manage, and manipulate virtually any type of data object. The Control Repository consists of a Common Access Interface and one or more data bases (see FIG. 17—Items 1 thru 5). These data bases may be:

- A Relational Data Base consisting of a collection of tables of data where the columns contain the attributes of related data and the rows are the instances of the data.
- An Object Oriented Data Base consisting of a collection of object instances of classes where the attributes are the class members.
- A Control File Data Base consisting of a collection of files where the records are the instances of data and the attributes are arranged along the records.
- A Directory Data Base consisting of a collection of file directories which may or may not contain files. Their relationships are described by the directory structure. The instances can be either sub-directories or files.

This repository communicates with users and the data repositories through a plurality of Managers, each performing a unique function. These Managers act as building

blocks which can be combined in numerous ways to support environments ranging from a small user community to a global enterprise.

Our preferred embodiment employs a relational database to serve as the Control Repository. Each data object in the Data Management System (DMS) is assigned a unique identifier that permits all information about the object to be recorded and tracked by a multiplicity of relational tables. The physical data is stored using conventional storage management techniques which allow any type of data (text or binary) to be tracked in its original form. The data may even reside on multiple platforms.

Users of the DMS communicate directly with the Control Repository, through a Communications Manager, to initiate some or all data management functions. Upon initiation, the Communication Manager employs one of the other Managers to complete the task. Our preferred embodiment contemplates the use of software service machines, known as Automated Library Machines, which execute requests on behalf of the users. These Automated Library Machines (ALMs) automatically enable the proper Manager to carry out the desired task, while freeing up the user's environment to perform other activities. The Communication Manager also enables the ALMs to communicate directly with the Control Repository.

In order to optimize data storage, our embodiment uses a PFVL paradigm to identify all data in the DMS by Package, File Type, (Data Type), Version and Level. Packages are arbitrary divisions of data whereby all the data has some common association. A Data or Package Manager defines the structure for the Package and performs all data management administrative functions. Levels are typically associated with "degrees of goodness" or quality. Data typically enters the DMS at low Levels with minimum entry criteria. As the quality improves, it is promoted to higher Levels until eventually being released as a finished product. Our system supports robust promotion criteria definitions which may exist for every PFVL in the DMS. Versions allow multiple variations of the same piece of data to be processed and managed simultaneously. One Version may be independent or based on another, which eliminates the need for common data to be repeated.

The present embodiment expands the PFVL paradigm into a means which enables the Data Manager to configure a Package under numerous structural arrangements. For example, the Data Manager may store all the data into a single physical repository, or segregate it by PFVL. The structure may contain multiple entry points, which enables data to be Fast-Pathed into non-entry Library Levels. This feature supports unlimited branching where any Level may have multiple lower Levels, each of which may have multiple lower Levels. Levels may be denoted Working Levels which constitute the minimum structure all data in a given Package and Version must traverse prior to release. Working Levels are transitory places where no data resides permanently. In addition, our embodiment permits the existence of Release Levels where data resides upon release as a finished product. These can be Regular Release Levels where data may only enter from the highest Working Level and remain permanently frozen. There is also a concept of a Sideways Release Level which serve as a repository for modifications made to data residing in Regular Release Levels.

In order to aid users and third party tools in locating data, our embodiment offers a Search Manager. The underlying utilities provide a means to search for data starting at a specified Level and Version. If the search fails to find the data at the starting location, it will traverse the structure

ascending Levels until all Levels in the current Version are exhausted. If the current Version is based on a previous Version, the search will traverse the previous Version. The search engine will locate data stored on multiple platforms and a single invocation can find multiple data objects of the identical or different data types. The Search Manager offers a multitude of options and features to seek out data in public and private Libraries, to sort and filter the results, and to perform the search with or without the assistance of the Control Repository.

Our preferred embodiment describes the most sophisticated form of the DMS which incorporates a Communication Manager to manage all communications with the Control Repository. It employs a series of communications machines capable of queuing and prioritizing queries initiated by users or Automated Library Machines. The mechanism enables unlimited access to the Repository regardless of the number of simultaneous queries supported by the database. The Communication Manager also provides a medium of information exchange for all other Managers and the ALMs. Since the Communication Manager supports multiple platforms, it acts as an agent to provide remote access to the Control Repository through conduits such as the Internet.

The present embodiment provides data control and security through a Lock Manager which offers three types of locks. First, there are Out for Update or Ownership locks which permit a user to check out a data object and modify it without fear of another user making a simultaneous update. Our embodiment also provides a means for transferring ownership of a piece of data from the primary owner to a designated surrogate without the primary owner's intervention. Upon completion of the transfer, the primary owner is automatically notified of the ownership transfer. Additionally, the preferred embodiment provides an environment where multiple users can own the same piece of data at different Library Levels.

In addition to ownership locks, the Lock Manager offers Move and Overlay locks which can be used to prevent data from being moved through the DMS or overlaid by the data at lower Levels. It also interacts with the Aggregation Manager to provide locking of entire an Bill of Materials, and it interacts with the Process Manager to provide an interlocking mechanism or data undergoing Automated Library Processing.

Our embodiment contains an Authority Manager to provide various types of user authorities down to the PFVL granularity. Interaction with the other Managers affords, but is not limited to, the following authorities:

- Data Promotion into and through public Libraries.
- Bill of Material Promotion through public Libraries.
- Creation of a Bill of Materials
- Setting the three types of locks on data objects
- Initiating Automated Library Processes
- Setting Level Independent Pseudo Process results

Our embodiment even permits pattern matching on the names of the data objects to add another Level of granularity beyond the PFVL.

In order to aid the Data Manager in performing the multitude of administrative tasks, our embodiment contemplates a Package Manager which includes utilities and user interfaces to accomplish the following:

- Set up Package Control Data such as Fix Management and P/N Control Levels.
- Define or dynamically reconfigure the Library Structure, including selection of data types to be tracked under the DMS.

Define the physical repositories of the data (down to the PFVL if so desired).

Balance workloads among Automated Library Machines.

Define, manage and edit:

Automated Library Processes

Authorities

Automated File Groups

The Package Manager supports Authority Profiles which permit the Data Manager to assign users to a classification and apply authorities to the entire classification. It also incorporates the concept of pre-defined process groups and templates which allow process definitions to be standardized across multiple packages. In our preferred embodiment, these definitions can be stored in flat files called Progroups or within a sample Package in the Control Repository. The Package Manager also offers a variety of report generators for information about installed Levels, Versions, data types, Automated Library Machines, process definitions, process results, authorities, fix management and release control information. Upon completion of all interactive editing, the Package Manager employs a batch commit process which converts the changes into a series of Control Repository modification instructions.

Our Data Management System also employs various utilities to aid in performance tuning and automated recovery of the Control Repository, data archiving, Control information back-up, and a mechanism to generate performance tuning reports.

Additionally the DMS employs a Library Manager to execute all data movement, check out, manipulation, check in and deletion. It also contains a Process Manager to provide Automated Library Processing and External Data Control. Also present is a Problem Fix/Part Number/Release Control Manager to associate and track problems and part numbers to data as well as coordinate releases. Finally an Aggregation Manager is included for creating and tracking arbitrary collections of data objects.

Structure and Search Manager

The present embodiment incorporates a robust concept which permits a data management structure capable of tracking a plurality of data objects governed under similar or disparate processes. The concept is based on a paradigm in which all objects can be classified by Package or its synonym Library, Type of Object (Our preferred embodiment denotes this as File Type), Version and Quality Level. This paradigm is hereafter referred to as the PFVL paradigm. (See FIG. 18—Items 1 thru 7). Under this arrangement, a Package is simply defined as a grouping of objects with common characteristics. In some environments, such as Chip Design or Software Development, a Package is referred to as a Library. Commonality may be defined in numerous ways. For example, all the components in a Library may be members of the same higher level component (such as all the ASICs on a PC Board), and thus may be considered a single Package. Another example may be all the programming modules written by a particular software development team.

Within a Package or Library, data is organized by Version. Versions allow parallel evolution of the same components to coexist in the same Library. For example, two Versions of a Video Graphics chip may be developed in tandem, one for the PCI interface and one for VL-BUS. Our embodiment allows the two flavors of design to use the same object names, reside in the same Library, and even be at the same Level, simultaneously.

For each Version, there is a Level Structure. In our preferred embodiment, Levels denote a degree of completeness, stability or quality control. The definition of

"degree of quality control" is left up to the Data Manager. Our embodiment simply affords the Data Manager a means to establish a Level structure commensurate with the goals and objectives of the user community.

All data objects are identified by name and type. Our preferred embodiment depicts all objects as files, but they can be any type of object that exists in a computer environment. The type of object serves as the fourth qualifier in the PFVL paradigm. In summary, an entity characterized by a single name may have multiple types of data objects, simultaneously residing in multiple Levels, of multiple Versions and spanning multiple Libraries.

In addition to denoting degrees of completeness, our embodiment permits Levels to be chained together to allow data to migrate from one Level to the next. Any or all of these Levels can be designated as Entry Levels whereby data may enter from a user's Private Library. Levels are also categorized as Working Levels or Release Levels. Data in Working Levels is transitory; and must eventually migrate to a Release Level. Release Levels serve as permanent storage vaults for a coherent set of data. Once the data is promoted into a Release Level, that Level is frozen and a new Release Level is opened. Data always migrates from the highest Working Level into the current, or open, Release Level. Any Working Level may be promoted to from another Working Level, or serve as an Entry Level for data coming from a Private Library. Release Levels are more restrictive. The current Release Level can be promoted to, but can't be an entry point for outside data. Frozen Release Levels are neither entry points nor are they promotable. Our embodiment does provide a means to thaw a frozen Release Level and delete data from it.

Our embodiment also discloses one special type of Release Level known as a Sideways Release Level. These Levels always branch out from a regular Release Level, but unlike regular Release Levels, data is permitted to enter from a Private Library. This arrangement permits updates and "fixes" to problems found with data residing in a frozen Release Level.

The PFVL structure lends itself to a powerful feature of the embodiment known as a Library Search Engine. In many commercial Data Management Systems, the means for establishing quality Levels often require physical segregation of data into a common repository. Usually this entails making copies of the data to multiple locations. Although our preferred embodiment will permit data to be copied as it migrates from one Level to the next, the default action is for the data to move to the higher Level. The Library Search Engine can be used to pick a starting location in the Library Structure and seek out a collection of coherent data objects, regardless of their current Library location or physical residence. The Search Engine and it's underlying algorithms are discussed in the Search Manager section.

Now, in considering our Control Repository illustrated by FIG. 17 and our Data Repository illustrated by FIG. 18 in implementing our system, an embodiment of a database that is used includes IBM's multimedia DB/2, or the databases of Informix which allow image and audio fields instead of plain text. With this kind of database, represented by the databases in the drawings, in the Control Repository, one could have image or audio fields. So a user could do a library search for all red sweaters that match a photograph of some hot new fashion design, at or above the P1 price level starting with the "Parisienne" version. (Hint: this scenario could be an implementation of our DMS in a large mail order clothing outlet which caters to Web shoppers.

One can use a text database such as DB/2 as the Control Repository, combined with the multimedia DB/2, or other

multimedia supporting database as a data repository. This would allow storage of the actual data, audio and images into DB/2 where the various pattern matching engines could be used, yet still allow the data to be "controlled" or managed using our techniques with the cheaper and simpler DB/2 (which really only needs simple textbased tables to work).

By using a multimedia database (e.g. DB/2 version) as the control repository, one could perform "queries" using voice commands like: "Display the Fix Management Data for Part 18F4475" or "Show me electrical checking results for all schematics at the Q1 level".

FIG. 19 illustrates an example Library Structure. To clarify the example, the overall structure is segregated by Library File Type with inverted tree 110 denoting the ASIC structure, and inverted tree 120 denoting the Firmware structure. To begin with, both trees have Working Levels WL1, VL1 and VL2 in common. These are known as the Default Levels and these would exist for all LFTs in the Library. Turning our attention to the ASIC structure, it has additional unique Levels known as WL2, WL3, CD1, CD2 and CD3. This type of arrangement could be used to accommodate high Level design being done at the Default Levels, synthesized parts being processed on the WL2-WL3 branch, and custom design being done at the CD1, CD2 and CD3 Levels. Although our embodiment permits data to enter into any of these Levels, the Data Manager controls the Entry Levels. In this example ASIC data may enter CD1, CD2, WL1 or WL2.

The highest Working Level is VL2, and above that is the current Release Level known as AR3. Above that are frozen Release Levels AR2 and AR1. AR1 is the original release of the ASIC design, and AR3 will contain the most recent. To the left of Release Level AR1 is Sideways Release Level ARP1. Additionally, Release Level AR2 has Sideways Release Levels ARP2 and ARP3. As stated above, when data enters any of the Release Levels except AR3, it is "trapped" and can't move to another Level. However, it can be located with the Library Search Engine.

Since there are 7 entry points (CD1, CD2, WL1, WL2, ARP1, ARP2, and ARP3), there are 7 independent search paths. The user may initiate a search for data at any point in any of these 7 paths. A search initiated at a Working Level or regular Release Level will move towards the "tree trunk" and up to the oldest Release Level (AR1). The search path for CD2 would be:

CD2→CD3→VL1→VL2→AR3→AR2→AR1
(terminator)

Searches beginning at a Sideways Release Level will migrate towards the "tree trunk" then turn upward towards the oldest Release Level. A search beginning at ARP3 would look like:

ARP3→ARP2→AR2→AR1 (terminator)

Turning our attention to inverted tree structure 120, this represents the Firmware tree. In addition to the Default Working Levels, this tree has Working Levels FD1 (which is an Entry Level) and FD2. It also shows Release Levels FR2 and FR1 (which is frozen). FR1 has one Sideways Release Level known as FRP1.

Further unique structures can exist for each LFT in the Library, or an LFT can use the Default Structure. In addition, any structure may be replicated to form multiple Versions. In this way a single Library is equipped to handle a multitude of data management tasks. The only restriction on the present embodiment is that any given Level in the tree may migrate to one and only one higher Level. For example, CD3 may not point to both VL1 and VL2.

The entire structure of every Library in the DMS is stored in tables within the Control Repository. These tables show information about each Level denoting attributes such as Entry Level, Promotable Level and the physical location of the repository. In order to improve performance and availability, our preferred embodiment permits this structural information to exist in an external file for quick reference by users running applications in their Private Libraries. An example of a structure file is shown in FIG. 20.

The structure file in FIG. 20 is divided into 6 sections. Each section contains the following four tokens:

The first token contains three pieces of information delimited by a / in our preferred embodiment. The / can be used to parse the first token as follows:

1. The LFT where XXX denotes ALL LFTs in the Library.
2. The Version where XX denotes ALL Versions in the Library.
3. The Source Level where 00 is a special keyword denoting any user's Private Library.

The second token is the Target Level

The third token is a Put/Promote flag which decodes as follows:

- NN Source Level not Puttable/No Promotion Path from Source to Target
- NY Source Level not Puttable/Promotion Path from Source to Target
- YN Source Level Puttable/No Promotion Path from Source to Target
- YY Source Level Puttable/Promotion Path from Source to Target

The name of the physical repository of the Target Level. This can be multiple tokens depending on the computer platform.

Making use of parts of an existing Cadence TDM system

Now having discussed PFVL as part of an AFS version, we note that among existing systems, Cadence does not have such an AFS version, but does provide DM software which can run on a Sun Microsystems Workstation. We have concluded that the current Cadence system is insufficient for our complex needs. However, Cadence has an effective underlying command line interface for the TDM function which drives all TDM functions. This command line interface can be modified so as to incorporate it into our methodology with an AFS environment.

Basically, our Data Management System needs to employ what we call the PFVL Paradigm.

Remember to optimize data storage we use a PFVL paradigm to identify all data in the DMS by Package, File Type, (Data Type), Version and Level. Packages are arbitrary divisions of data whereby all the data has some common association. The PFVL acronym was derived from IBM internal jargon, but the same principles can be applied using Cadence parlance as;

Library—Variance—Quality Level—View—Cell—Version if our PFVL structure and principles are adopted, as they should be. The PFVL structure and process provides that every piece of data in the Data Management System (DMS), regardless of origin or importance, is tracked by PFVL. In other words which our PFVL structure and principles are adopted in a Cadence system every piece of the design, whether it's a schematic, piece of VHDL, a layout, or documentation which is associated to a

Library—Variance—Quality Level—View—Cell—Version has all of the data associated so that the system ensures every piece of data has the PFVL (here 6 attributes) associated with it.

Furthermore, our DM principles state that all data and control information is tracked in an architecturally centralized location consisting of a Data and Control Repository. An "architecturally centralized location" does not require that all the data must be kept in a single Unix directory, nor that all the control information must reside in a single metadata file. Nor does it imply the whole system must be governed by a single database. What it says is that the user must perceive the system in a manner by which all data appears to be tracked uniformly. So, an example might be that I have a design for an MPEG decoder. The physical design is done in Cadence so the actual layout data physically resides in a "Cadence style directory structure". However, we have FrameMaker documentation which explains and diagrams the physical design, but as FrameMaker documentation is done outside of Cadence, it is not stored originally as Cadence data. Physically this is kept in a completely different Unix directory, maybe completely isolated by system and location from any "Cadence data". Using our system, however, wherever the documentation is located, the DMS still tracks both data objects by

Library—Variance—Quality Level—View—Cell—Version which enables the user to do things like find/view all the data associated with the MPEG even if multiple pieces of data are in physically disparate locations. The reason this works is that the system ensures every piece of data has these 6 attributes associated with it. The control repository can also be distributed as long as each component follows the structure and process of the architecture. For example, the Cadence data to most quickly integrate our structure and process of our architecture into a Cadence system, one would use a routine for tracking the data by TDM using TDM's control files to act as the Control Repository for the physical design of the MPEG decoder. Likewise, all FrameMaker documentation might be tracked by a Lotus Notes Database so that it can be made available to both designers and external folks simultaneously. As long as the TDM Control Files and the Lotus Notes Database adhere to the PFVL architecture, the user is hidden from the inner workings. All he knows is that he runs some front-end script or GUI menu where he can ask to find all the MPEG decoder information under a given cell name at a particular Quality Level of a particular Variance in a certain Library. The DMS looks through the various physical Control Repositories, finds the layout and FrameMaker views, finds their exact Version numbers and locates them in the proper physical data repository.

Our PFVL structure and process architecture should be used in combination, with many other useful Data Management features which we have developed and explained. We feel the following features should tie universally implemented in combination with our PFVL structure:

Using the PFVL architecture to set up a single logical Data Management system for design data (Cadence and non-Cadence). Various parts of the design are tracked in separate shared libraries. Each library would consist of N Quality Levels and M Variances (N and M are determined by each Data Manager based on the type of information stored in that library).

A dynamic Bill of Materials Tracker would exist to allow PD, Timing and Simulation Coordinators (Integrators) to easily identify all the desired pieces of a design at a particular Library, Level and Variance to be built into a "model". Once integrated into the model, the BOM Tracker would monitor the actual versions of the data objects and

alert the Integrator if any versions become obsolete. The BOM Tracker can also be used to perform promotions of cohesive units of data between levels.

Automated Library Processing whereby tools, checks, and automated tasks can be launched either during movement of data between levels or while data is stable within a level. Results would be associated to the exact data objects used in the process (via the PFVL architecture) and retained in the Control Repository. These results can serve as promotion criteria to ensure data is promoted only when it has achieved the desired level of quality. The Data Manager would be able to "program" his library to run any available Library Processes either in a particular sequence or in parallel.

External Data Control whereby results obtained from tools run outside of the DMS can be securely incorporated into the DMS with the same data integrity as an Automated Library Process initiated from within the DMS:

A Locking mechanism which not only performs simple Check-Out, Check-in to assert ownership, but allows ownership by PFVL. So, two different designers could check out different versions of the MPEG decoder at different Quality Levels. An example might be that the primary designer has the decoder checked out at the lowest library level, but the PD Integrator finds a minor electrical problem at the highest level which is causing a DRC check to fail. He simply has to insert a buffer so he goes ahead and checks out that level of the design, makes the change and checks it back in. He can continue with the DRC run while he informs the primary designer of the required change for the lower levels. This locking mechanism would also permit surrogate ownership of the same piece or design, such that two people would work on the same piece together. The system automatically ensures only one has it checked out at a time, but allows the other to "take ownership" if he needs to. Automatic notification and complete history tracking reduce the risk of miscommunication among the partners. In addition the locking mechanism would also support other types of locks such as Move and Overlay whereby coherent sets of data could be temporarily frozen into a Quality Level to prevent accidental movement or obsolescence while a lengthy model build is underway.

Problem Fix Management and Engineering Change Tracking would provide various utilities to ensure that fixes to problems are contained within the proper EC. In addition certain information about the fix would be tracked in the Control Repository to enable various types of escape analysis, status reporting, etc. Mechanisms would exist to prevent or minimize the risk of the same piece of design being associated with multiple ECs or a piece of design being attached to the wrong EC.

There are many other features of our preferred embodiments which we could employ, but we need to implement these concepts and we have working algorithms which we have provided in these applications which are suitable for use with TDM and some of TDM's existing features, like policies, can be employed as part of a foundation for our system. However all our algorithms expect the Control and Data Repositories to follow the PFVL architecture. Hence implementation needs to ensure use of things like hierarchical projects to properly emulate Quality Levels. We also need to ensure that Variances can be supported with the current TDM architecture. In order to use the systems together, at some point a conversion needs to provide the following mapping:

Library	Variance	Quality Level	View	CellName	Version	(Cadence)
to	/\	/\	/\	/\	/\	
	/\	/\	/\	/\	/\	
Package	Version	Quality Level	Type	FileName	Iteration	(IBM DMS)

In other words we need to be able to use the TDM API to make queries about data using the above mapping, between PFVL attributes, regardless of name or origin, as the data may originate in IBM DMS, Cadence, ViewLogic (see below) or elsewhere.

Both private and public libraries need to be provided. With respect to private libraries Cadence employs Working Areas (a limited kind of Private Library) in which a user can define one or more private libraries, each of which can reside in any physical AFS location desired by the user. All authorities are done through AFS, so the user controls who can access or update their private areas. There is a GUI which makes it easy for a user to set up a Working Area and create private libraries. TDM includes a special Integration Area (a limited kind of Public Library) which is designed specifically for people performing coordination or verification tasks. These areas are similar to the regular working areas in terms of how they are defined, but these have additional functionality which is further discussed with regard to our Library Manager. Private libraries are a function of our own library management system. With regard to public libraries, Cadence's TDM uses a Project Management tool to assist the Data Manager in various DM tasks such as defining and maintaining public libraries. The DM may set up an unlimited number of Release Areas, but the system is limited and only permits one Integration Area per public library. Furthermore, the DM also has no control over the initial physical location of the data (in the Integration Area or Release Areas), and the system automatically imposes a single directory tree structure for the public library. We note that within AIX is a function called "permissions", and when TDM could be run in an AIX environment, when data must be relocated to another physical repository, there is a TDM command to perform the function. In such an environment, since all authorizations would be done via an AIX permission, the DM would have complete control over file access, write authority, and data removal.

There are several significant problems with this TDM system area of public and private libraries. The current TDM system fails to differentiate between a Working Area, an Engineering Area, a Level Area and a Release Area. In complex development projects, these are different and each of these area must be differentiated by any concurrent engineering staff. TDM makes it impossible for the DM to define any type of library structure with multiples levels and/or versions. One could try to change the use of a Release Area to mimic an engineering level, but the required amount of daily iteration combined with the required degree of parallelism makes this impractical in a commercial environment. Integration Areas may be considered as having adequate characteristics to serve as a model for engineering levels, and data can be continually promoted into the Integration Area until such time that the Integrator feels it should be released. However, TDM is restricted to a single Integration Area per public library. This may be compared to our own public library methodology which permits multiple Integration Areas per public library. Without this function, along with others which we may note, which needs to be implemented in the system as we have described it, TDM would not satisfy necessary requirements of a complex system.

A complex system requires the ability to define data types and perform nongraphical DM tasks. We note TDM offers a powerful selection of data management commands which enable a Data Manager to do virtually an entire job from outside of the Cadence TDM system. TDM's command line API interface allows, for instance, any existing BOM application to run entirely in a system command line environment. This allows for integration with nongraphical tools of third parties. This also provides the basis for data managers to write unlimited utilities (via C programs, shell scripts, perl scripts, etc.) to automate or enhance their productivity. But we note that Data Types must be selected from a list of master cell views administered on a project wide basis. It is not evident that users can create a unique new Data Type themselves, even though they can create new data and select from existing Data Types (cell views). Apparently, also, non-Cadence data can't be tracked by an existing Library Browser. Library processing, particularly private library task launching is an aspect of our own development. In this regard, Cadence offers the concept of Policies to permit task launching against any data object in the DMS. With proper arguments, we believe that any executable code could be run from a policy. This would permit a user to call a system function, third party tool, shell script, perl script, C program, etc. from within a policy. For conversion of an existing piece of code to a policy, an appropriate API code would be required.

TDM has commands which can be invoked from a command line, within a script, program, or policy. This flexibility in the policy architecture of the tools when coupled with the set of TDM data management commands, does present the opportunity to combine our own developments with this aspect of TDM. While TDM invokes a policy from a TDM session, it is possible to launch a policy from an AIX window. To incorporate our developments, there would need to be tight integration of policy launching from a Library Browser like that we have in our library management function discussed below (although we note that Cadence provides a library browser without our functionality).

We note that policies can be chained together such that a policy sequence can be run. However, the sequence in TDM is not determined by the Data Manager or user, but rather by sorting the policies in alphabetical order by name, then using that as the order of execution. Policies can be configured to launch during other DM operations, such as Check Out or Check In, or they can run stand-alone.

Public library task launching is required. In this regard, TDM provides that all policy functions are available to a Working Area (b, Private Library) and also available to data in an Integration Area or Release Area (Public Library). The existence of a fully functional command line interface in TDM allows the Data Manager or Integrators to perform a variety of tasks, without invoking a separate Cadence tool call Framework.

Cadence also has a separate tool product PCS which allows one to graphically diagram a process flow. The user simply clicks on the various process boxes, and the tool then executes underlying policies. This provides a nice graphical interface to assist designers who prefer working with GUIs.

Basic design tasks and library promotion is handled by our development's library management system and Library Browser. In this regard, TDM uses a Working Area -to->Integration Area -to->Release Area paradigm as their basic design control flow. By mapping between their model and our model of designer library -to->working library levels -to->release levels, there is a close correspondence. The Cadence submission facility corresponds to a promotion utility for moving data from a Working Area to the Integration Area, and from the Integration Area to the Release Area. The TDM front end allows one to perform Check Out, Check In, Promotion, Switching work areas, and viewing Integration areas. However, the user interface for the Working area is inadequate, and does not serve as a Library Browser replacement.

The latest 4.4 Version of TDM is not capable of checking data out of public libraries, checking in, promoting to integration and release areas, displaying and browsing data in integration and release areas and easily switching between multiple working areas. In the Cadence framework, all designer functions need to be executed from the library browser, and the subset of data management commands used on a daily basis should be supplied, as this is an absolute requirement for our complex design systems environment.

The Cadence system does not, as does our development, supply multiple "integration areas" and our complex design systems environment does require multiple working library levels. Apparently, standard opinion is that one could use a shared integration area in order to keep a complex part, e.g. an E-Unit in the same library. What has not been appreciated with an Element and System Sim sharing the same integration area, is that there is invariably a phase shift in build frequency. This makes the assumption of this prior work impractical, for they should not and do not always co-exist. Thus there is a need for our multiple integration areas and these need to be properly addressed to successfully implement a complex design control system.

Furthermore, our library management system, as we discuss, provides for extracting data from a public library. In our methodology both the integrators/coordinators/model builders and the designers need simple access methods to data in release areas.

The prior art has treated releases as a static configuration. Thus, the Cadence TDM concept of a Release is a complete set of pointers to each component of an entire design, something akin to a Bill of materials, a static configuration. Release Areas always contain pointers to every piece of design. This makes model builds in the single Integration Area very straight forward since the integrator need only point to a Release as a starting point, then incrementally add in designer's updates and fixes. To do this the integrator must use a separate tool suite, which is awkward. But, as we have discovered, the designer(s) needs to access released data. The absence of such function would be catastrophic in a complex system design environment. With the current Cadence product, the designer is forced to use the TDM front-end to perform basic data management functions such as switching/viewing work areas, checking data in, checking data out, promoting data, and accessing integration and release areas (public libraries). In fact, in the Cadence system, it is easy for a user to accidentally point to and edit a design's control data, instead of the design itself. The risk of designer error thus is unacceptable.

This is solved by our development by providing a way to sort data by cell name, or cell view, with a tree structure display, categories, etc. A user has, and should have, no access to control and meta data. Use of a prior art system

which could require a designer to ping-pong between an unsatisfactory library browser and a rudimentary front-end is risky and unacceptable.

Nevertheless, in fairness, we would say that we could run typical design DM tasks non-graphically with the set of TDM commands coupled with a command line interface to permit any of the basic design functions (Check out, check in, promote, view work areas, view release areas) with the capabilities of an operating system like AIX. From an AIX command line, using script, or any type of executable program, this part of DM can be performed in such an environment.

Our system provides a way for sharing/transferring data ownership. In this system, file locking mechanisms are present. Cadence's TDM offers two methods for locking files during checkouts. The first is an exclusive lock (the default) where check out of any version of a design component renders all other versions of the same component unavailable for editing. There is a problem with this solution, in that it not only prevents a second designer from updating someone's components, it also prevents the original owner from working on two versions of the component simultaneously. For example, if a bug is found in the Element and System sim models, and they contain two different versions of a component, the designer must check out, fix, and check in each version sequentially.

Cadence also has a method using non-exclusive locks, whereby two different users may check out different versions of a design simultaneously. Again there is a problem with this solution. Since the system won't allow the same user to hold two non-exclusive locks on the same piece of data, in the aforementioned example of an Element and System sim design bug, sequential fixes are still required (unless the original designer gets a second user to fix one version of the design while he fixed the other, sometimes a problem when the original designer is in the USA and the second designer is in Japan or Germany, France, England, Canada or one or more of many other countries, to indicate a real possibility). Furthermore, there is a possibility that a second user can grab and modify a different version of the design without the original owner ever knowing about it. The result is a system which has no utility for transferring ownership, as we have provided, making it difficult to operate on different versions of the design in parallel as required for concurrent engineering. Furthermore, depending on the type of locking employed, with TDM it may not be possible for another designer to act as a surrogate for the original designer in an emergency situation (i.e. Where the original designer has the file checked out and has to unexpectedly take a leave of absence). The TDM system does not permit the DM to override or reset a designer's check out locks.

Our system, we will note here, has a locking mechanism which enables transfer of ownership permanently or temporarily, and allows for an override or reset of the check out locks, and provides notification to the original designer or administrator in the event of check out.

A system requirement in this kind of system is a Bill of Materials (BOM) mechanism which should have a satisfactory way to create BOMs, as we have provided. The Cadence Checkpoint Manger provides a nice mechanism to create Checkpoints (BQMs). Checkpoints are manageable data objects, which means that they can be checked in, checked out, promoted and tasks can be run against their members, the Cadence system permits Checkpoints to contain other Checkpoints as members, thus allowing hierarchical creation. All Checkpoint information is stored in an

ASC file, which, coupled by the TDM command line API interface, permits the TDM to interact with third party tools.

Our preferred embodiment, it will be noted permits locking a Bill of Materials. In this regard, we note the TDM makes it possible to save all versions of a design component and use pointers to denote which version is associated to the Integration Area or a Release Area. However, it appears possible to delete a version, and no mechanism exists for the Coordinator to "lock down" the versions of his model to prevent this. No mechanism appears to exist to prevent newer versions from entering the Level at which the Coordinator is working, therefore he can't always be assured of working with the most recent data. However, with knowledge of the way we provide for locking a Bill of Materials, one could modify the Cadence TDM paradigm to accommodate lock down. A coordinator would first build some type of model, and create a Checkpoint to track the contents of the model. After the model is working properly, it could on a later day be declared a success. During the time the Coordinator first ensures none of the member; of the model disappear (are deleted or overlayed) and ensures that they can't be accidentally promoted to another level. Secondly, the Coordinator would be enabled to know if, on the later day, if any of the members of the model have become back-level. This may be achieved by a Cadence "policy" solution, where a policy code is written to set a "freeze" flag against each member of a checkpoint. This would be satisfactory.

One of the features present in Cadence's Checkpoint Manager is that the BOM can be static (which means the contents are a snapshot version numbers) or dynamic (which means the contents change to always include the most recent version of a component). Adding or deleting members of an existing BOM is relatively easy. We also update members of a BOM in accordance with our preferred embodiment.

It will be noted that we have addressed the issue of gathering BOM status or real-time BOM invalidation, something not possible with such tools as provided by Cadence. Furthermore, we can use the BOM as a handle to promote the updates into an Integration or Release area from the Working Area. In addition, we provide "real-time" notification if members are deleted or modified. This capitalized on the ease in updating and deleting members in the BOM.

It will be noted that we provide utilities for examining BOM operations. In the Cadence system, the owner can delete the Checkpoint without affecting the data objects, but there is no convenient method for viewing the status of a BOM or its members.

We have provided a convenient set of utilities for use an examining BOM operations. We provide a method for viewing the status of a BOM or its members. With our provisions, in a TDM system, not only could an owner delete the Checkpoint without affecting the data object, but task launching could be built in for BOM members. To launch tasks in a cadence system, a Policy could be written to loop though the members of the BOM and either launch tasks against them or determine their current version is most recent. Our method for BOM movement through a library is a substantial advance. Cadence's TDM has no BOM promotion mechanism.

We have described how we provide for BOM movement through a Library. Our ideas could be implemented in a Cadence like system, when our ideas for multiple integration levels are incorporated, by using the underlying Cadence Checkpoint mechanism to move BOMs through the Library levels.

We have provided for program fix management, another substantial advance. TDM has no built-in fix management function.

Cadence policies can be used to achieve our problem fix management functions. The mechanisms which we use to implement the tasks should be used.

Our release manager enables making design changes for subsequent releases. The TDM model of moving Integration -to->Release area supports initial releases and sequential ECs well. Subsequent releases are constructed using any current release as a base and the sole Integrator has complete control over the Integration -to->Release Area path. The Data Manager can define any number of release areas, control access, and has complete freedom of nomenclature.

Cadence's TDM lacks any inherent concept of multiple level structures within a given project. Although Integration Areas can serve as a single level, it does not permit the establishment of separate engineering and release levels with the ability to physically segregate the data accordingly. Also, in this connection, we note that the Data Manager can't define any physical location of released data; and TDM automatically assigns new Release Areas to a directory hanging off of a top-level directory for the entire Library. Data can be displaced by a relocate function. Everything for project management is sequential. With the Cadence system, multiple versions cannot satisfy the parallel developments required for concurrent engineering.

As we would note, our release manager allows the creation of multiple "Integration Areas" making it possible to run multiple ECs or Design versions in parallel. We can define the physical location of released data. This is important in large volume design components which require extensive ECs.

With our release manager, ECing a released component is handled. This task resolves around a designer accessing a released piece of design, modifying it, and then releasing it under a new EC. With TDM a designer via the user interface can open a Release Area and check out a piece of data into any Working Area. This allows a designer some flexibility in setting up multiple Working Areas or private libraries for different ECs. However, the users are subjected to the failings of the TDM user interface. Most frequent DM problems occur when a designer mistakenly sends a component into the wrong EC stream. While with TDM, where only one EC is processed at a time, one could minimize this problem by having a single Integration area for a Public Library, but this requires the Integrator to ensure that he selects the appropriate target Release Area and handle the Integration Area -to->Release Area promotion himself. Mixing components into the wrong EC stream is possible and risky.

With regard to ECing a released component, multiple Integration Areas could be provided and coupled to manage data from multiple ECs. It is possible to iterate and verify parts with Multiple Integration areas. However, one still has to properly implement the DMS to avoid having a part from different ECs mixing together, so our Design Control System if building models needs to be implemented.

In our Design Control System we allow building models from released data. This tasks requires Integrators to be able to easily access all components relating to a Release, and perform tasks such as netlisting a release. TDM has a single Integration area, and the Release is a pointer to a complete set of design components, but there is no library browser support since some model build task may require launching tools from within the framework. TDM has no support for multiple Integration areas, making it impossible to perform multiple model builds in parallel. We discuss how to fix design problems in multiple releases within our system. It is not possible with TDM to make simultaneous fixes. A user

is unable to Check out multiple versions of the same piece of design, even though the Release area structure does permit a Data Manager to establish area for "patches" to Released data (i.e. design patches for the test floor).

With the Design Control System it is possible to conduct an ISO approved verification audit, Cadence has no way of conducting an ISO approved verification audit as a task, so adoption of our process management function and our release manager methods needs to be employed.

An ISO approved verification audit requires a ISO Quality Record. This task requires a DMS capable of storing results from tasks along with the proper pedigree information for the files used to run the tasks. It can then be enhanced to produce output in a comparable format to the ISO Quality Record. In this connection we use our process management function and our release manager.

Making use of parts of an existing ViewLogic system

As we will discuss various changes which need to be adopted for existing software in order to make use of part of that software in our new environment, we note that ViewLogic provides software tools for Unix type workstations, such as the preferred AIX version which is capable of running in an AFS environment. Again, we believe that the current ViewLogic software, which is now insufficient for our complex needs, needs to be changed.

ViewLogic's View Data is based on ASC files which can be edited and viewed and easily maintained. Nevertheless, our PFVL structure and principles need to be adopted, whether aspects are called

Package—Version—Quality Level—Type—FileName—Iteration (IBM DMS) or

Library—Variance—Quality Level—View—Cell—Version

or some other name, version, level, type, filename, iteration structure if our PFVL structure and principles are adopted, as they should be. The PFVL structure and process provides that every piece of data in the Data Management System (DMS), regardless of origin or importance, is tracked by PFVL. In other words which our PFVL structure and principles are adopted in a Cadence system every piece of the design, whether it's a schematic, piece of VHDL, a layout, or documentation which is associated to a

Package—Version—Quality Level—Type—FileName—Iteration (IBM DMS) or

Library—Variance—Quality Level—View—Cell—Version

or some other name, version, level, type, filename, iteration structure has all of the data associated so that the system ensures every piece of data has these 6 attributes associated with it.

Here we would recommend ViewLogic adopt our DMS including

(a) our DM principles state that all data and control information is tracked in an architecturally centralized location consisting of a Data and Control Repository;

(b) using the PFVL architecture to set up a single logical Data Management system for design data (ViewLogic and non-ViewLogic);

(c) providing a dynamic Bill of Materials Tracker to allow PD, Timing and Simulation Coordinators (Integrators) to easily identify all the desired pieces of a design at a particular Library, Level and Variance to be built into a "model", etc.;

(d) providing Automated Library Processing whereby tools, checks, automated tasks can be launched either during movement of data between levels or while data is stable within a level and where results would be associated to the

exact data objects used in the process (via the PFVL architecture) and retained in the Control Repository;

(e) providing External Data Control whereby results obtained from tools run outside of the DMS can be securely incorporated into the DMS with the same data integrity as an Automated Library Process initiated from within the DMS;

(f) providing a Locking mechanism which not only performs simple Check-Out, Check-in to assert ownership, but allows ownership by PFVL, so, two different designers could check out different versions of the piece of a design element at different Quality Levels; and

(g) providing Problem Fix Management and Engineering Change Tracking which would provide various utilities to ensure that fixes to problems are contained within the proper EC.

Now, when one reviews the ViewLogic product, we can with our own perspective, interpret the ViewLogic Working Areas as able to serve as a limited-kind of Private Libraries. Users must first be assigned to the "Team" by the Data Manager. Once they exist on a Team, they may create as many Working Areas as they desire, and they can locate them anywhere in the directory structure of an AFS system. AIX permissions serve as the only means of authorization, so the user has complete control over who has access to their data. The Working Area is always driven by the existence or absence of the physical files. This means data can always be created or deleted from outside of ViewLogic's ViewData, and the Working area is guaranteed to provide an accurate image.

Unfortunately, while it is easy for a user to create multiple Working Areas, the ViewData DMS has an awkward user interface for managing multiple areas. For example, the user can't display all the files in multiple areas at the same time. They must switch between Working Areas via a Set Environment function, which does not provide adequate visual cues to assist the user in knowing where they are currently pointing to. This makes it difficult to use and adjust to.

The ViewLogic's ViewData uses a Team paradigm to perform shared data management. This is somewhat analogous to a Public Library. Each Team may have any number of Release Areas, and members of a Team. Each member may have unlimited Working Areas. The DM can create Release Areas, specify the physical location of the data, and rename or delete the Release Area at any time. While ViewData can be flexible in creating Release Areas, the architecture does not differentiate between levels. A DM can't define the type of structure required for concurrent engineering with multiple levels. Promoting data between Release Areas is not adequately addressed. Furthermore, the architecture requires the user to only permit the user to look at data in one Working Area and one Release Area at a time. There is no way to look at data in all Release Areas simultaneously. All data is stored in a Vault during a Check-In operation. Since data can be checked in while residing in a Working Area, or during a desired promote to a Release Area, the Library View imposes the restriction that a user must use the Set Environment function to point to a Working Area and one Release Area. The view refreshes to show a union of all the data in those two repositories. If the user then wants to see data in a different Release Area, they must again use the Set Environment. Since there is no way for the user to look at all Release Areas simultaneously, even the simplest DM maintenance and debug tasks (which are easy in our system) in ViewData are a virtual nightmare. Furthermore, the concept of unionizing data between a Working Area and a single Release Area causes problems which is solved by our system.

ViewData could be modified through the use of an RCS version Segregation to create "virtual" levels and Versions according to our teaching. This would be very similar to how we use static configurations today as virtual Levels.

ViewLogic's Data Management functions can be performed from within their GUI or by updating ASC files which hold all the Library Configuration information. The ViewData DMS allows one to work with data imported from outside the ViewLogic environment. We have copied a Cadence symbol from one of our designs into a ViewLogic subdirectory serving as a Working Area. By simply refreshing the Library View, our Cadence symbol appeared as a new data object with a visual cue indicating it is in an unmanaged state. Thus we have shown that the existing ViewData tools allow the end users to have the power to define new data types by simply creating them or copying them into their Working Areas. This can be done via the GUI and by selecting the Data Type from a list of Data Types defined by the Data Manager for the Team.

With regard to our private library task launching part of our preferred embodiment, we note that we could use ViewLogic scripting language known as ViewScript to provide for any command line activity to be launched from the Library Viewer. A built-in API permits tasks to be chained together, and ViewScripts can be written to test the results of one task before proceeding with the next task. All task launching is controlled by a single ASC file which may reside in a centralized location for a project, or each user may have their own. Within this file, the user may use the API to launch a task such as MTI Compile a stand-alone view script which may be simple or compiles, and imbed actual ViewScript code to do more complex things such as chaining tasks together with dependency. Once the file is saved, the new task appears in a task menu within the Library Viewer. It is simple enough that users of ordinary programming skill can launch bottom line commands in accordance with our modification.

With regard to our public library task launching we would note that all the task functions available to a private library should also be available to a public library. However, our experience is that ViewLogic does not provide any such function, although, theoretically they must know how to enable this function. However, ViewLogic has no distinct command line interface, and therefore the graphical ViewData product must be invoked momentarily even if the intent of the user is to launch a task which can be run non-graphically, such as a CLSI compile.

With regard to our Design Control System's basic design tasks and library promotion ViewLogic's View Data is not suitable for concurrent engineering in which: designers only perform a limited amount of verification before sending data to a public or shared library;

- designers frequently need to work with back-level data;
- people performing verification frequently require shared access to all parts of a design;

- designers may have to work on multiple versions of the design in parallel;

- designers may "own" many design components;

- designers may share pieces of the same design (i.e. A logic designer writes VHDL, and a circuit designer creates layout), and

- designers may belong to multiple "Teams". Within ViewData's design environment a user tends to work on an isolated piece of design, belonging to a single Team. In this paradigm the user tends to reside in the same Work Area and iterates within their private library until the

design is stable enough to be "released" into the public library. This paradigm assumes an orderly sequential design flow where the user is only interested in working within the latest version of design, and rarely needs to access back-level data. The designer is assumed able to perform all the necessary task with little need for data sharing. Within this limited structure, the ViewData system provides the necessary tools to edit, update, verify and release the design.

However, when faced with a concurrent engineering approach, many of the needed characteristics simply don't exist in the ViewData product. For example, once a designer performs a Set Environment to point to the proper Working Area, it's possible to browse and edit any data residing in that Working Area; however, if a designer needs to work with data in a different Working Area, he must switch the environment. This makes it impossible to see all of a designer's data simultaneously.

The ViewData system requires the designer to Check Out a piece of data in order to edit it. This is the correct implementation, however, the system further employs the restriction that data must be checked out (owned) by the designer to simply browse the data. This is true whether the design is an isolated text file (like VHDL), or has hierarchical dependencies (schematics). Basically the user must check out all components of a schematic in order to properly browse the schematic. This makes it difficult and impossible to share pieces of design or employ scenarios where multiple people anywhere need to debug a problem.

ViewData's library browser uses a pair of icons to represent the state of a data object. The icons clearly indicate whether the current version of an object is managed or unmanaged, and whether it's up to date or obsolete. The system has built in safeguards to prevent a back level piece of data from accidentally being promoted. However, in our concurrent engineering approach, we solve the need to promote back-level data at required times, so the lack of this capability in ViewData's design is a detriment of this prior attempt.

ViewLogic's Work Area approach imposes constraints on design which are unacceptable from a concurrent engineering point of view. Data is automatically sorted by class (Data TYPE) as opposed to design component names. There is no way to re-sort the data. As a preferred concurrent engineering approach useable under our design control system can and should be based on working with all types of a given component, use of a ViewLogic system to implement this approach would become cumbersome for designers owning many pieces. The visual clues provided by the ViewLogic system for indicating which Work Area a user is currently pointing to are insufficient. If a user has multiple Work Areas, it's too easy to begin working in the wrong area and not realize it. Furthermore, the Set Environment can switch back to a default setting without the user knowing a switch occurred. This creates numerous problems if the user is in the midst of editing a check out piece of data, and then checks it into a wrong Work Area by accident. An when the designer is unaware, a concurrent user cannot be aware, and so could later check out the piece of data placed in a wrong Work Area by accident, and further compound the problem.

Our concept of promoting data from a Private to a Public Library in the ViewLogic environment has similar problems. For example, the user must be pointing to a proper Work Area/Release Area union (using the Set Environment function) BEFORE initiating a "promote". The act of promoting a file in ViewData is called a Check In and Release, which consists of simply clicking on a menu pick. Since

there is no user interaction, failure to set the proper Work Area/Release Area union results in a promote where data is either taken from the wrong source, deposited in the wrong repository, or both. The user may never know this occurred. There is no means to perform an automated hierarchical promote where the user points to a top level schematic, and the DM system automatically traverses the design to gather all the instantiated components.

Another major issue making ViewData an unacceptable substitute to our Design Control System is the absence of a method for promoting data between Library Levels (Release Areas). The ViewLogic DM system does not support this concept in a suitable manner for integrated design control. In ViewData, promotion of all data from Level A to Level B requires a user to check out all the data into a Working Area, switch environment to a new Level, and perform the Check In and Release function. In addition to being cumbersome this is not possible under certain situations: Since check out implies acquiring ownership, this means the Data Manager must be able to get ownership of all the data at Level A. If a designer in a concurrent engineering environment is in the middle of an update, this is not possible with ViewData.

ViewLogic needs to have the ability to sort the Library Viewer by design component, something now absent in the ViewLogic system, yet one which we have provided.

We provide for extracting data from a Public Library; however, the ViewLogic DMS has no adequate substitute function. In the best case, with ViewLogic, the user must perform a Set Environment function and establish a proper Work Area/Release Area union before any data can be extracted from a Public Library. Thus a user needs to have knowledge of which Work Area was used as a source of promotion into the Release Area, something not assumable in an integrated concurrent engineering environment. As a single data object promotes through multiple Release Areas, it gets more difficult to understand the association. The Library View can display all existing versions of an object, but nothing in the system indicates which version is currently residing in which Release Area. Furthermore, the only type of sorting is by Data Type, so all levels of a schematic are mixed together. In short, the only means a user has of extracting data from a Level, is to perform the Set Environment, Click on Check Out and hope for the best. In many instances, this may work properly, but when data exists in multiple Levels, this means possible severe data integrity risks.

Our integrated design control system for concurrent engineering allows a user to work with back level data (e.g. for fast-pathing sim fixes). When one understands this and can compare it with the ViewLogic system, it will be appreciated that that system has major limitations. For example, if a Version 1.5 is checked into a Level 1, and a Version 1.6 is checked into Level 2, any attempt to then work with Version 1.5 results in difficulty. The system default is to encourage a user to work with 1.6 because it assumes the user is mistakenly grabbing an obsolete version.

Any improvement of the ViewLogic system in connection with extracting data from a Public Library would need to address permanent storage. ViewData has no method to delete unwanted versions from permanent storage. Thus, in the event a user makes a mistake and promotes data to a wrong Level, there is virtually no way to clean it up properly. Also, as the project matures, and numerous versions materialize, the Data Managers request some way to delete the old unwanted data to reclaim space and clean up libraries. This is a problem with ViewData that needs to be addressed to adopt our preferred process.

We provide for sharing/transferring of data ownership. Within the ViewLogic system the only means for formal transfer of ownership is the act of one user checking a file in, followed by a second user checking it out. This system thus is not adequate for concurrent engineering because it relies on personal communication and coordination. Furthermore, suppose, as happens, two users have different versions of the same design component checked out for edit simultaneously. Because one was not aware of the other, until it comes time for one user to promote his modification into a Public Library, a problem will arise. However, even when the user has promoted his modification into a Public Library, that user does know that someone has another version check out, but is unable to find out which other version and who has it locked out. This is not a trivial problem, because in a concurrent engineering environment different users constantly need to update the same piece of design at different levels without any knowledge of another's actions. The result is a loss of data integrity.

We provide a built-in utility to transfer ownership, notify one user that another requests ownership, and provide a facility to monitor who has various versions of a design checked out at any point of time. Our solution provides a foreman mechanism allowing multiple versions of a data object to be updated simultaneously, with multiple ownerships and notification being handled as appropriate for the task.

We note that running ViewLogic DM functions or tasks can be performed nongraphically in a way accommodating our own design control system, making changes to follow our preferred embodiments possible.

Regarding a comparison of the viewLogic system, with the manner we employ for creating a bill of materials. ViewLogic DMS offers the concept of a Collection and a Checkpoint. The difference is that a Checkpoint is a static snapshot of a coherent aggregate of data objects identified by their exact version numbers, analogous to our Bill of Materials. A ViewLogic Collection is a grouping of data objects where each object in the group is always the latest version. The purpose of a Collection is to act as a "handle" by which the user can perform tasks on the entire group with a single invocation. This concept is similar to our File Groups.

The ViewLogic primary utility for creating and modifying either type of aggregate is the Collection Editor. This tool is easy to use, and enables one to easily create Collections and Checkpoints as well as add and delete members. This includes creating hierarchical Collections and Checkpoints. Despite this advanced function in the ViewLogic system, it poses problems in trying to achieve BOM tracking in an environment like ours. The ViewLogic system requires the user to check out or "own" all members they wish to include in the Collection or Checkpoint. Consider the ease where an Element Simulator wants to build a BOM containing all members in a sim model. The Checkpoint would appear, upon a first impression, to be a solution, however, the Element Simulator must first check out all the versions of data in the model. This immediately leads to:

- (1) the need for the Element Simulator to have enough AFS space in his Working Area to hold the snapshot of data, something not easily addressed; and
- (2) the problem of data integrity previously mentioned because of the lack of any way for multiple versions to be updated simultaneously, as the designer is most likely needing to work with back-level versions of data.

Given this severe limitation, the possible power of Collections and Checkpoints, is inadequate for a concurrent

engineering approach like that of our integrated design control system.

The Viewlogic BOM handling needs to have an ability to expand a Checkpoint in a Library Viewer and utilize the icons as a visual cue to represent the BOM status. These would be modifications that could be employed within VIEWLOGIC to make this area satisfactory.

We provide the ability to examine BOM operations/utilities. ViewLogic, in this regard, allows management of Collections and Checkpoints data objects in their own right, as they are afforded the same treatment as regular data objects. For example, Collections and Checkpoints each can be Checked Out, Checked in, Promoted (with some difficulty), and Deleted, and ViewScripts can be written to run tasks against their members. Thus functions can be performed against THE BOM without adversely affecting the data objects being tracked by the BOM. The iconized representation of the BOM members in the Library View allows the user to obtain the current status.

However, BOM movement through the library, like that we provide, is severely limited in the ViewLogic system, so our methods should be adopted. We do acknowledge that it is possible to promote a Collection, but with difficulty. After repeated tries, we have learned that if one accepts the restriction that a user must "own" all members of a Checkpoint, then one is able to promote an entire BOM from a Private Library into a Public Library. However, since the problems we discussed with respect to Level-to-Level promotion of regular data objects, which also pertain to BOMS, exist, there is no ability to move large BOMs easily from a Private Library into a Public Library. This is a real shortcoming of the ViewLogic system and a compelling reason we believe of adopting our total system.

While design tasks can be performed non-graphically in the ViewLogic system, it is not possible to perform BOM tasks non-graphically. While ViewScripts can be written to perform many functions involving Collections and Checkpoints, they require ViewLogic ViewData to be running in a graphical environment. There are a number of disadvantages to such a system, illustrating the need for our system's ability to perform BOM tasks nongraphically. For example, if a timing model can be created using a non-graphical timing tool, it should be able to interact with a BOM tracker non-graphically. We provide for a command line environment, and it is important for a BOM tracker, as we provide (see the Section 4.6 discussion), to coexist in the same environment as the tool it is intended to work with. This is not possible with the ViewLogic system.

There is a need for the problem fix management as we provide. ViewLogic DM has no built in fix management functions.

Some fix management functions with the ability of ViewLogic to execute specially written ViewScripts could be achieved. There was a proposal to address these concerns.

It must be easy in an integrated design control system to make design changes for subsequent releases. When one tries to track an equivalent of our functions onto the ViewLogic system design difficulties arise and the effort becomes difficult, even though at the highest level it is possible to make design changes for subsequent releases. We have achieved this result with unnecessary difficulty because we determined that because there is no type of Working or Engineering Level and one is forced to use the Release Levels for design iteration in the ViewLogic system. This requires one to perform Level-to-level promotions, which have the problems previously mentioned. One then designates a specific one of the Release Areas as a Release Level.

In order to make a design change, that data must be referenced. To do this one could create an overall system level Library and create a schematic instantiating components from the other Teams. Upon librarying that schematic, is is still difficult to work with the combined set of data. Because of the Working Area I Release Area Union paradigm forced by the system, a designer is unable to see all of the data, comprising the overall schematic, simultaneously. Instead, the designer must repeatedly use the Set Environment function to cycle through the various Working Area/Release Area combinations. Also, no data can be browsed in the public areas. All data has also to be checked out into the designer's Working Area, just to look at it. Finally, working with multiple Teams is cumbersome, even though the Data Manager has complete control over establishing the Release Areas, including their physical locations, and no Library Search mechanism is necessary, since the release area which has been designated as a Release Level consists of a complete set of pointers to the components used in the overall design. While making it possible to achieve the base result, this system is unnecessarily difficult and not conducive to concurrent engineering in an integrated design control system.

One of the aspects of desirable concurrent engineering would be the ability of the system to EC a released component, particularly when there is the ability to release control if multiple ECs are being processed in parallel without risking that data can be intermixed between the ECs. The problem with the paradigm used by ViewLogic is that an EC stream is represented by a Working Area/Release Area Union. We have noted that there is a lack of visual cues indicating the current union, and a possible promotion mechanism with no user verification of the source and destination. There is also no means to delete a version if it enters the wrong storage vault, making it impossible overall to minimize the risk of components being released into the wrong stream, something that must be avoided.

With regard to building models from released data, akin to the method we discuss, the ViewLogic system provides that a Release Area contains a full set of pointer to the entire design; however, a model builder must check out all the data into his Working Area. This is inefficient use of AFS space, especially if multiple models are to be built in parallel, since to achieve this task the user must set up multiple Working Areas to use as model build areas.

View Logic allows the Data Manager to establish necessary Release Areas to house "patches" to released data (i.e. design data patches for the test floor); however, the act of creating and releasing such patches often entails a designer working on multiple versions of a design component in parallel. The current ViewLogic system has many problems related to multiple check outs of the same component and working with back-level data. We would refer by way of contrast to the sets we use in fixing design problems in multiple releases.

ISO approved verification audits, which we discuss under Section 6.6, requires a DMS capable of storing results from tasks along with the proper pedigree information for the files used to run the tasks. It can then be enhanced to produce output in a comparable format to an ISO Quality Record. The current ViewLogic system offers no such mechanism, and our process management function and our release manager methods needs to be employed. While we have described our preferred embodiments of our inventions, it will be understood that those skilled in the art, both now and in the future, may make various improvements and enhancements which fall within the scope of the claims which

follow. These claims should be construed to maintain the proper protection for the inventions first disclosed.

What is claimed is:

1. A method for computerized design automation comprising,
 - accessing a file and database management system for managing a plurality of projects as a design control system,
 - organizing data repository for each project for data records and a control repository comprising a common access interface and one or more databases,
 - said control repository communicating with users of said design control system for fulfilling requests of a user and the data repositories of said data management control system through a plurality of managers, each manager performing a unique function; and
 - combining selected ones of said managers for supporting an computerized design automation application environment suitable for multiple users of a user community located at workstations anywhere in the world accessible via a network, an intranet, extranet or via the internet; and
 - defining for each project a control repository and one or more data repositories to store, manage and manipulate any data object, whereby said control repository communicates with users and the data repositories in numerous ways to support environments ranging from a small user community to a global enterprise; and
 - tracking all data and control information in an architecturally centralized location consisting of said data repository and said control repository using a single logical PFVL paradigm to identify all data in the DMS by Package, File Type, (Data Type), Version and Level, and wherein packages are arbitrary divisions of data, whereby all the data has some common association to a library.
2. A method for computerized design automation according to claim 1 wherein every piece of the design which is associated to a

component data element to be developed in tandem while using the same object name and residing in the same library and at the same level simultaneously.

- 5 5. A method for computerized design automation according to claim 4 wherein for each said version, there is a level structure denoting a degree of completeness, stability or quality control enabling said data manager a means to establish a level structure commensurate with the goals and objectives of the user community.
- 10 6. A method for computerized design automation according to claim 5 wherein at least some of the data objects are at a level chained to another level to allow data to migrate from one Level to the next, and wherein any or all of these Levels can be designated as Entry Levels allowing data to be entered into this Entry Level from a user's Private Library; and wherein there are also levels categorized as working levels and release levels wherein data in working levels is transitory, and must eventually migrate to a release level, while release levels provide permanent storage vaults for a coherent set of data.
- 15 7. A method for computerized design automation according to claim 1 further comprising, providing an architecturally centralized location which does not require that the location be physically in the same location but requires that the user must perceive the system in a manner by which all data appears to be tracked uniformly which enables the user to do things like find/view all the data associated with the design when multiple pieces of data are in physically disparate locations.
- 20 8. A method for computerized design automation according to claim 1 further comprising, the steps of mapping in order to use the system together with other systems the system attribute for conversion to provide the following mapping:

Library	-	Variance	-	Quality Level	-	View	-	CellName	-	Version
\\		\\		\\		\\		\\		\\
Package	-	Version	-	Quality Level	-	Type	-	FileName	-	Iteration.
\\		\\		\\		\\		\\		\\

45

Library—Variance—Quality Level—View—Cell—Version has all of the data associated so that the system ensures every piece of data has all its PFVL attributes associated with it, permitting every piece of data or attribute of a design element in the Data Management System (DMS), regardless of origin or importance, to be tracked by its PFVL single logical association of every piece of the design.

3. A method for computerized design automation according to claim 1 further comprising, providing that data management model structure capable of tracking a plurality of data objects governed under similar or disparate processes, wherein all objects are classified as part of a library, having one or more types, each type having one or more versions, and each version having one or more levels.
4. A method for computerized design automation according to claim 3 wherein said library is a grouping of objects which all have common characteristics causing them to belong to the same library grouping, and wherein within a library, data is organized by version, wherein versions allow parallel evolution of the same component data element to coexist in the same library enabling multiple versions of a

9. A method for computerized design automation according to claim 1 further comprising, providing that the system provides working areas which serve as a private library for a user assigned to a team by a data manager, and to perform shared data management each team may have any number of Release or Integration Areas, while team members may have unlimited Working Areas and the system data manager can create release or integration areas, specify the physical location of the data, and rename or delete the Release or Integration Area at any time, and create virtual levels to define the type of structure required for concurrent engineering with multiple levels and variances, promote data between virtual levels and into Release Areas, and enable users to look at data in all Release or Integration Areas simultaneously.

10. A method for computerized design automation according to claim 1 further comprising, enabling with said PFVL single logical paradigm attributes of non-system originated data to be tracked and made accessible along with system originated data with a system command.

11. A method for computerized design automation according to claim 1 further comprising, enabling a user call for a

system function, third party tool, shell script, perl script, C program, or any other type of application programming interface from within a user system may be invoked from a command line.

12. A method for computerized design automation according to claim 1 further comprising, enabling a locking mechanism which enables transfer of ownership permanently or temporarily of PFVL data, and allows for an override or rest of the check out locks, and provides notification to the original designer or administrator in the event of check out.

13. A method for computerized design automation according to claim 1 further comprising, providing that all data and control information is tracked in an architecturally centralized location consisting of a data and control repository for a project using a single logical PFVL paradigm, and wherein said system provides a dynamic Bill of Materials Tracker to identify all the desired pieces of a design at a particular Library, Level and Variance to be built into a "model".

14. A method for computerized design automation according to claim 1 further comprising, providing that all data and control information is tracked in an architecturally centralized location consisting of a data and control repository for a project using a single logical PFVL paradigm, and wherein said system provides Automated Library Processing whereby tools, checks and automated tasks can be launched either during movement of data between levels or while data is stable within a level and where results would be associated to the exact data objects used in the process and retained in the Control Repository.

15. A method for computerized design automation according to claim 1 further comprising, providing that all data and control information is tracked in an architecturally centralized location consisting of a data and control repository for a project using a single logical PFVL paradigm, and wherein said system provides Problem Fix Management and Engineering Change Tracking which would provide various utilities to ensure that fixes to problems are contained within the proper release or EC.

16. A method for computerized design automation according to claim 1 further comprising, enabling interaction with said Managers of the DMS to perform tasks such as Automated Library Processing, Problem Fix and EC management, BOM Tracking, Authority and Lock checking before, during and after the promotion of data in the DMS.

17. A method for computerized design automation according to claim 1 further comprising, enabling the use of Automated Library Machines in a client server environment for purposes of processing promotion requests, initiating and executing Automated Library Processes, installing newly created data into the DMS, performing any other functions provided by said Managers of the DMS, and permitting

execution of non-DMS tasks through the use of an Auto-reader virtual queue.

18. A method for computerized design automation according to claim 1 further comprising, enabling accessible multiple working library levels, and means for extracting data from a public library.

19. A method for computerized design automation according to claim 5 all data objects are identified by name and type.

20. A method for computerized design automation according to claim 19 wherein at least some of the identified data objects are identified by a type of data object which depicts the objects as files.

21. A method for computerized design automation according to claim 19 wherein at least some of the identified data objects are identified by a type of data object which depicts the objects as files, while other data objects identified by the same file name exist, such that an entity of said data management system characterized by a single name may have multiple types of data objects, simultaneously residing in multiple Levels, of multiple versions and spanning multiple libraries.

22. A method for computerized design automation according to claim 6 wherein at least some of the data objects when the data is promoted into a release level, that Level is frozen and a new release level is opened such that data always migrates from the highest Working Level into the current, or open, Release Level.

23. A method for computerized design automation according to claim 6 wherein at least some of the data objects when the data is promoted into a release level, that Level is frozen and a new release level is opened such that any working level may be promoted to from another working level, or serve as an entry level for data coming from a Private Library while a current Release Level can be promoted to, but can't be an entry point for outside data and frozen Release Levels are neither entry points nor are they promotable.

24. A method for computerized design automation according to claim 1 further comprising, providing that every piece of data or attribute of a design element in the Data Management System (DMS), regardless of origin or importance, is tracked by its PFVL single logical association of every piece of the design.

25. A method for computerized design automation according to claim 1 wherein a database for the control repository includes a multimedia database.

26. A method for computerized design automation according to claim 1 wherein a database for the data repository includes a multimedia database.

* * * * *